VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA
OSTRAVA
EKONOMICKÁ FAKULTA

KATEDRA SYSTÉMOVÉHO INŽENÝRSTVÍ A INFORMATIKY

Návrh a tvorba internetové aplikace běžící v reálném čase

Design and Development of Real-time Web Application

Student: Jaroslav Klimčík

Vedoucí diplomové práce: Ing. Pochyla Martin, Ph.D.

Ostrava 2015

VŠB - Technická univerzita Ostrava
Ekonomická fakulta
Katedra systémového inženýrství

Zadání diplomové práce

Student: **Bc. Jaroslav Klimčík**

Studijní program: N6209 Systémové inženýrství a informatika

Studijní obor: 6209T025 Systémové inženýrství a informatika

Téma: Návrh a tvorba internetové aplikace běžící v reálném čase
Design and Development of a Real-time Web Application

Zásady pro vypracování:

1. Úvod
2. Teoretická východiska tvorby internetové aplikace
3. Analýza a návrh řešení internetové aplikace v reálném čase
4. Realizace internetové aplikace
5. Závěr

Seznam použité literatury
Seznam zkratek
Prohlášení o využití výsledků diplomové práce
Seznam příloh
Přílohy

Seznam doporučené odborné literatury:

CANTELON, M., M. HARTER, T. HOLOWAYCHUK and N. RAJLICH. *Node.js, MongoDB, and AngularJS Web Development.* New York, USA: Manning Publications, 2013. 416 p. ISBN 978-1617290572.
DAYLEY, Brad. *Node.js, MongoDB, and AngularJS Web Development.* Indiana, USA: Addison-Wesley Professional, 2014. 696 p. ISBN 978-0321995780.
ROHIT, Rai. *Socket.IO Real-time Web Application Development.* Birmingham: Packt Publishing, 2013. 140 p. ISBN 978-1782160786.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Pochyla, Ph.D.**

Datum zadání: 21.11.2014
Datum odevzdání: 25.04.2015

doc. Ing. Jana Hančlová, CSc.
vedoucí katedry

prof. Dr. Ing. Dana Dluhošová
děkanka fakulty

Prohlašuji, že jsem celou práci, včetně příloh, vypracoval samostatně.

.....

Jaroslav Klimčík

V Ostravě dne 25.4.2015

Rád bych poděkoval vedoucímu práce panu Ing. Martinu Pochylovi, Ph.D. za jeho cenné rady, vstřícnost a čas, který mi byl z jeho strany poskytnut. Chtěl bych také poděkovat i těm, kteří mi jakkoliv pomohli při zpracování této diplomové práce.

Obsah

1. Úvod

V roce 1995 byly internetové prohlížeče velmi odlišné na rozdíl od těch dnešních prohlížečů. World Wide Web existoval teprve čtyři roky, avšak vykázal znaky takové popularity, že se o něj začaly zajímat i některé známější společnosti, jako byla například společnost Netscape. Ta stála za vznikem slavného prohlížeče Netscape Navigator a pro jeho druhou verzi se vedení rozhodlo vložit do něj programovací jazyk. Tento úkol padl na softwarového inženýra Brandena Eicha, který tak dal vzniknout počátek nového programovacího jazyka JavaScript.

Po mnoho let byl JavaScript považován jako podřadný jazyk sloužící pouze pro amatéry na poli vývoje internetových stránek. Poté byl však představen AJAX, a když Google zveřejnil své webové aplikace Gmail a Google Maps, bylo všem náhle jasné, že technologie AJAX může změnit webové stránky na webové aplikace. To také inspirovalo novou generaci webových vývojářů, kteří posunuli JavaScript na další úroveň.

Z počátku začaly vznikat především různé knihovny třetích stran, jako je jQuery nebo Prototype. Poté se však do vývoje vložil i samotný Google díky jeho prohlížeče Google Chrome, který je poháněn V8 JavaScriptovým enginem a rapidně tak vzrost výkon JavaScriptu. Následoval vznik nových platforem, jako je například Node.js umožňující spouštět JavaScript na straně serveru, databáze jako MongoDB rozšířily využití JSON formátu, a frontendové frameworky jako je AngularJS se díky moderním prohlížečům mohly stát velmi účinným nástrojem pro tvorbu komplexních webových aplikací. Téměř po dvaceti letech vývoje je tak JavaScript prakticky všude. Jazyk, který byl považován jen pro amatéry, schopný vykonávat pouze malé části kódu, se nyní stal jedním z nejpopulárnějších programovacích jazyků.

V současnosti existuje celá řada JavaScriptových frameworku tvořící dohromady takzvané „stacky“, nejznámější a nejvyužívanější je však otevřený a modulární MEAN stack. Ten využívá MongoDB jako databázi, Express jako backendový webový framework, AngularJS jako frontendový framework a Node.js jako platformu. Kombinace těchto komponent zajišťuje potřebnou flexibilitu při vývoji moderních webových aplikací.

Cílem této diplomové práce je tvorba internetové aplikace běžící v reálném čase s využitím MEAN stacku společně s modulem Socket.io. Aplikace bude umožňovat uživatelům se zaregistrovat pomocí svého osobního účtu, nebo pro přihlášení využít jiných

sociálních služeb, jako je Facebook, Google nebo Twitter. Aplikace bude sloužit pro vkládání inzerátů, respektive pro vkládání nabídek a poptávek při hledání partnera pro společné sportování, kdy po přidání nového inzerátu jej okamžitě uvidí i všichni ostatní uživatelé webové aplikace. Tyto inzeráty pak budou kategorizovány podle typu sportu a lokality a podle toho je vyhledávat a třídit. Bude také umožňovat přidávat si jiné uživatele jako své přátelé a psát jim zprávy, které si mohou přečíst hned bez nutnosti obnovovat stránku. Pro správce bude obsahovat přívětivou administraci pro správu uživatelů, inzerátů a sportů. Zároveň bude splňovat všechny funkční i nefunkční požadavky klienta. Práce bude využívat moderních technologií s využitím principů REST architektury a poskytovat jednoduché API.

2. Teoretická východiska tvorby aplikace

2.1 HTML

HyperText Markup Language, běžně používaná zkratka HTML, je programovací jazyk navržen pro tvorbu internetových stránek. Tyto webové stránky si pak může prohlížet kdokoliv, kdo je připojený k Internetu. Pro HTML se neustále vytvářejí nové revize a pokračuje se ve vývoji, aby splnily požadavky stále se rozrůstajícího počtu internetových uživatelů.

Za jeho vytvořením stojí fyzik Tim Berners-Lee, pracovník evropské výzkumné organizace CERN, který v roce 1989 vytvořil hypertextový systém založený na Internetu. Později pak Berners-Lee specifikovat HTML a vytvořil pro něj prohlížeč a server obsluhující požadavky klienta. Roku 1991 publikoval první dokumentaci popisující prvotní jednoduché elementy jazyka. Na počátku roku 1994 vznikly první verze HTML a HTML+ a o rok později organizace Internet Engineering Task Force vytvořila pracovní skupinu HTML Working Group, která dokončila verzi HTML 2.0, první HTML specifikaci s úmyslem vytvořit standard. (Berners-Lee, 1995) Další vývoj pod záštitou IETF byl zastaven konkurenčními zájmy a od roku 1996, s příchodem komerčních dodavatelů softwaru, je specifikace HTML udržována konsorciem World Wide Web Consortium, zkráceně W3C. V roce 1999 byla publikována verze HTML 4.01, která se na dlouhou dobu stala oficiálně uznávaným standardem. V roce 2004 začal vývoj na nové verzi HTML5, na které se podílela také komunita WHATWG, která se ke konsorciu W3C přidala roku 2008, a verzi dokončila a standardizovala 28. října 2014.

2.2 HTML 5

HTML5 je nejnovější hypertextový značkovací jazyk pro tvorbu internetových stránek od konsorcia World Wide Web Consortium, zkráceně W3C. První návrh byl zveřejněn již v roce 2008, ale až do roku 2011 se s ním nijak více nepracovalo. Poté vznikla první předběžná a již použitelná verze HTML5, kterou programátoři začali používat, a společně začaly vznikat články o ní. Nicméně v té době byla podpora v různých webových prohlížečích velmi slabá,

avšak v současnosti všechny hlavní webové prohlížeče (Chrome, Safari, Firefox, Opera a Internet Explorer) nabízejí podporu HTML5.

HTML5 pracuje s CSS3 a je stále ve vývoji. První finální, kompletní a stabilní specifikace byla vydána v říjnu 2014 a poté se její hlavní jádro a funkce o něco rozšířily. (Sikos, 2014) Některé části HTML5, které byly odebrány z původní HTML5 specifikace, se odděleně standardizovaly jako samostatné moduly (ve formě knihoven, které může programátor využívat) jako například Microdata nebo Canvas. Technické specifikace jako například Polygot Markup byly také standardizovány jako moduly. Některé W3C specifikace které byly původně odděleny, se nakonec připojily k HTML5 jako její rozšíření, například SVG. Vytvoření dalších specifických funkcí by jen zpomalilo standardizování první stabilní verze HTML5 a proto budou k dispozici až v dalších verzích HTML5.1, HTML5.2 atd. Očekává se, že nejbližší verze 5.1 bude dokončena v roce 2016, v současné době probíhá její normalizace. (Buckler, 2012)

HTML5 je nástupcem dlouho používané verze HTML 4.01, která vznikla již roku 1999. Od té doby se internet znatelně změnil, a proto bylo třeba vytvořit novou verzi, která bude splňovat stále náročnější požadavky uživatelů a vývojářů. Nový značkovací jazyk byl vyvinut na základě předem stanovených norem:

- nové funkce by měly být na základě HTML, CSS, DOM a JavaScript,
- je nutné snížit potřebu externích pluginů (jako například Flash),
- zachytávání chyb by mělo být jednodušší než v předchozí verzi,
- skriptování musí být nahrazeno větším značkováním,
- HTML5 by měla být nezávislá na používaném zařízení,
- vývojový proces by měl být veřejný. (Vasile, 2013)

Nová verze byla vyvinuta za účelem jednoduššího a logičtějšího procesu kódování. Unikátní a působivé funkce HTML5 se týkají především multimédií. Mnoho funkcí vznikly především za účelem, aby uživatelé byli schopni spustit náročný obsah i na mobilních zařízeních, především na mobilech a tabletech, které mají mnohem menší výkon než klasické stolní počítače nebo notebooky. Syntatické funkce zahrnují nově elementy <video>, <audio> a <canvas>, ale také integraci vektorové grafiky. To značí, že multimediální a grafický obsah internetových stránek může být zpracován a vykonán mnohem snadněji a rychleji, aniž by bylo nutné využívat jiných zásuvných modulů nebo API.

V HTML5 existuje také mnoho nových elementů ovlivňující strukturu dokumentu, jako například <article>, <nav>, <section> atd. Nové tagy také ne vždy fungují stejně jako v předchozí verzi. Například značky záhlaví a zápatí označují nejen začátek a konec stránky, ale také začátek a konec každého jednotlivého oddílu. To znamená, že tyto dvě značky mohou být použity více než jednou v celé stránce.

I když HTML5 podporují všechny hlavní internetové prohlížeče, velkým úskalím je stále to, že každý prohlížeč využívá jiné renderovací jádro, například Google Chrome a Opera využívá jádro Blink, Mozilla jádro Gecko, Internet Explorer používá Trident a Safari nebo Android prohlížeč jádro WebKit. To vede k tomu, že ne všechny prohlížeče podporují všechny elementy HTML5.

2.3 CSS

V devadesátých letech minulého století si začali HTML kodéři uvědomovat, že píšou neustále stejné značky znovu a znovu na stejné stránce, což vedlo k mnohem větším HTML souborům a rostla také časová spotřeba a frustrace. CSS, česky kaskádové styly, bylo poprvé navrženo Håkon Wium Liem v říjnu 1994, který v té době pracoval v CERNu s Timem Berners-Leem. Účelem CSS bylo mít jeden soubor, který definuje všechny hodnoty, a všechny stránky kontrolují tento soubor a podle toho formátují své stránky. V důsledku toho lze mnoho formátovacích značek v HTML přenechat CSS souboru a využívat pouze strukturální prvky, například záhlaví, paragrafy, odkazy apod., a oddělit tak strukturu od grafické prezentace. Kdykoliv pak je třeba změnit vzhled internetové stránky, stačí pouze změnit jediný CSS soubor a všechny HTML stránky, které na daný kaskádový styl odkazují, se zobrazí se změněnou grafikou. Údržba webové grafiky je tak o mnoho lehčí.

Jak rostla HTML komunita, začaly se i rozšiřovat stylistické schopnosti stránek, aby se splnily požadavky webových vývojářů a web designerů. Tento vývoj však dal designerům i větší možnosti jak bude stránka vypadat za cenu komplexnějších HTML stránek. Začaly se objevovat různé druhy implementací v prohlížečích a uživatelé měli menší kontrolu nad tím, jak se zobrazí obsah webových stránek. Zlepšení možností webové prezentace bylo předmětem zájmu mnoha internetových komunit a bylo navrženo devět kaskádových jazyků. (Lie, 2005) Z těchto devíti návrhů měly především dva jazyky vliv na budoucí vývoj CSS: Cascading HTML Style Sheets za jehož vývojem stál Håkon Wium Lie a Stream-based Style

Sheet Proposal (SPP), jehož tvůrcem byl Bert Bos. Později spolu Lie a Bos spolupracovali na vývoji standardu CSS.

Jako záštitu nad CSS převzala skupina CSS Working group, která začala řešit nedostatky první verze CSS level 1 a vytvořila vylepšenou verzi CSS level 2, kterou oficiálně zveřejnila v květnu roku 1998. O rok později začal vývoj třetí generace CSS3.

Hlavním rozdílem mezi CSS2 a CSS3 je separace do modulů. Zatímco v předešlé verzi bylo všechno v jedné velké specifikaci definující různé funkce, CSS3 je rozdělena do několika dokumentů, zvané jako moduly. Každý samostatný modul má nové možnosti, aniž by byla ohrožena kompatibilita předešlé stabilní verze. V současnosti existuje více než padesát modulů, čtyři z nich jsou však zveřejněny jako oficiální doporučení:

- Media Queries,
- jmenné prostory,
- Selectors Level 3,
- barva.

Media Queries jsou velmi důležitým modulem CSS3. Umožňují totiž velmi jednoduše za určitých podmínek aplikovat různé styly, takže je možné obsah upravit na různé šířky rozlišení. Media Queries umožňují vývojářům přizpůsobit internetové stránky různým rozlišením, aniž bychom museli změnit nebo odstranit obsah. Mezi další důležité designové aspekty CSS3 patří také například ohraničení, které lze vytvořit zakulacené, aniž by bylo třeba nějakých „hacků“, čímž velmi pomohl web designerům a vývojářům. Dalšími zajímavými moduly jsou pak například přechody barev, stíny textu nebo mít obrázek jako ohraničení okolo prvku.

Z počátku zveřejnění CSS3 bylo, a někdy je potřeba dodnes, pro jejich použití potřeba tzv. vendor prefixes, které pomáhaly prohlížečům správně interpretovat kód. V současnosti hlavní internetové prohlížeče využívají tři různé vendor prefixes:

- -moz- pro Firefox,
- -webkit- pro Chrome, Safari a Operu,
- -ms- pro Internet Explorer.

Příchod CSS3 přinesl také mnoho nových pseudo tříd, včetně těch konstrukčních, které ovlivňují prvky na základě jejich pozice v dokumentu nebo na základě vztahů s jinými prvky. Mezi nejpoužívanější patří například :nth-child(n), který využívá numerické hodnoty (n) a ovlivňuje podřazené elementy ve vztahu k jejich postavení s rodičovským elementem, nebo

:first-of-type, který se zaměřuje na první specifický typ rodičovského elementu, jeho opakem je pak :last-of-type.

2.3.1 LESS

CSS má jednu nevýhodu, a to že je statické. To znamená, že musíme znovu a znovu opakovat kód v souboru s kaskádovými styly, čímž se stěží dodržuje princip DRY, neboli „Don't Repeat Yourself". Tomu se dá předejít použitím LESS, který využívá dynamických kaskádových stylů. Co dělá LESS dynamický, je to, že rozšiřuje CSS o proměnné, mixiny, operace a funkce a CSS se tak chová více jako programovací jazyk. Klíčovým prvkem pak je to, že zachovává původní používanou CSS syntaxi. (Gerchev, 2012) Ve větších webových aplikacích se údržba kaskádových stylů může velmi rychle stát chaotickou záležitostí, protože existuje mnoho stále se opakujících elementů a není v klasickém případě CSS možné využít jednoduché znovu použitelnosti.

Technicky vzato, LESS je CSS preprocesor. Předzpracování CSS metoda rozšiřující funkce CSS, kdy se prvně vytvoří soubor s kaskádovými styly v novém, rozšířeném jazyce, a poté se kód zkompiluje do tradiční CSS syntaxe, která je již čitelná pro internetové prohlížeče. LESS je jeden ze tří nejpopulárnějších CSS preprocesorů, společně se Sass a Stylus.

Proměnné v LESS mohou být deklarovány kdekoliv v kaskádových stylech a pak použity kdekoliv v CSS souboru. Změna jediné proměnné pak aktualizuje všechny části, kde se daná proměnné využívá. Proměnné lze také kombinovat a proměnné obsahující numerické hodnoty, jako jsou například pixely nebo barvy, lze sčítat, odečítat, násobit či dělit mezi sebou nebo s konstantními hodnotami. Dalším aspektem LESS jsou mixins, které umožňují znovu použít část elementu s jeho hodnotou v jakémkoliv CSS selektoru. Do mixins je možné přidat jakýkoliv počet objektů a také mohou přijímat libovolný počet parametrů. Často se používají při využití vendor prefixů abychom zabránili jejich všudypřítomným opakováním se a udělali tak LESS soubor lépe čitelným. (Potvin, 2014) V neposlední řadě velkou výhodou LESS preprocesoru jsou funkce, které se většinou využívají pro manipulaci s barvami nebo numerickými hodnotami. Tím se dá jednoduše vyhnout konstantního výsledku v jedné operaci. Příkladem může být potřeba použít na jednom místě určitou barvu a na druhém místě použít tutéž barvu, avšak o 50% světlejší. Toho lze v LESS dosáhnout velmi jednoduchým kódem:

```
@standard-background-color: #001ED5;

.StandardBackground {
   color: @standard-background-color;
}

.LightBackground {
   background-color: lighten(@standard-background-color, 50%);
}
```

Funkce `lighten` doslova zesvětlí barvu o procentuální hodnotu. LESS nabízí další užitečné funkce, jako jsou dědičnost selektorů, využití podmínek if/else, jmenné prostory, jednoduchého importování jiného LESS souboru, JavaScriptového vyhodnocení atd.

2.3.2 Twitter Bootstrap

Bootstrap je open source projekt, původně vytvořený pracovníky Twitteru Markem Ottem a Jacobem Thorntonem, umožňující tvorbu responzivních mobilních internetových stránek. Bootstrap obsahuje standardní sadu tříd, která umožňuje vývojářům vytvářet rychle webové aplikace jednoduše škálovatelné pro zařízení všech rozměrů a zahrnout běžné komponenty jako jsou například dialogová okna nebo validaci. Twitter Bootstrap se tak stal de facto standardem pro web design. (Harrison, 2014)

Bootstrap je účinný CSS front-end framework, který obsahuje HTML a CSS design na základě běžných uživatelských komponent jako jsou typografie, formuláře, tlačítka, tabulky, navigace, rozbalovací nabídky, upozornění, záložky, carousel a mnoho dalšího, stejně jako možnost využít některých JavaScriptových rozšíření.

Výhodou Twitter Bootstrap je i využití jeho API, které je poměrně jednoduché ho implementovat. Dalšími výhodami jsou například:

- ušetření mnoha času – s předpřipravenými šablonami a třídami je mnohem lehčí se soustředit pouze na vývojové práce,
- responzivní funkce – internetové stránky se správně zobrazí na zařízeních s různou šířkou bez nutnosti měnit HTML značky,
- konzistentní design – všechny Bootstrap komponenty sdílejí stejné šablony a styly, takže design i layout je během vývoje stále konzistentní,
- snadné použití – pro jeho použití stačí pouze základní znalosti HTML a CSS,

- kompatibilita s prohlížeči – Bootstrap je podporován všemi hlavními moderními prohlížeči, jako jsou Mozilla Firefox, Google Chrome, Safari, Opera a Internet Explorer,
- open source – je zcela zdarma a volně k použití s možností si projekt „forknout“, tedy vytvořit si alternativní větev programu, která je vyvíjená nezávisle, a dále upravit podle své potřeby.

2.4 JavaScript

JavaScript vznikl v roce 1995 za účelem zlepšení chování webových stránek, především pak internetových formulářů. S postupným vývojem Internetu byla podpora pro různé webové aplikace velmi důležitá. JavaScript byl navržen Brendanem Eichem a dále vyvinut společností Netscape.

Původní název byl LiveScript, avšak o něco později byl přejmenován na JavaScript jako marketingový tah na nový jazyk Java od společnosti Sun Microsystems, který se stal velmi oblíbeným a tím pádem i ziskovějším. Ačkoliv mají tyto dva programovací jazyky jen velmi málo společného, díky jejich názvu občas vznikaly zmatky s jejich příbuzností. (Flanagan, 2006) O několik měsíců firma Microsoft vydala kompatibilní verzi jazyka s názvem JScript společně se svým prohlížečem IE 3. Netscape na to předložil návrh JavaScriptu Ecma International, evropské normalizační organizaci, který vyústil v prvním vydání standardu ECMAScript v roce 1997. Standard pak roku 1999 získal významnou aktualizaci jako ECMAScript edition 3 a od té doby zůstal do značné míry stabilní. Čtvrtá verze nebyla dokončena díky neshodám ohledně složitosti jazyka. Mnoho částí čtvrté verze poté tvořily základ ECMAScript edition 5, zveřejněné v prosinci 2009.

Původní požadované vlastnosti JavaScriptu zahrnovaly zvýšení interakce mezi uživateli a webovými stránkami a asynchronní komunikace s prohlížečem, například online formuláře nebo flashové interakce. Tento dynamický programovací jazyk byl implementován do všech moderních prohlížečů a také do KJS, Rhino, SpiderMonkey, V8, WebKit, Carakan nebo do Chakra. JavaScript jakožto webový skriptovací jazyk je široce používán díky jeho dynamickým funkcím a využívá se především jako pro vytváření aplikací na straně klienta společně s některými dalšími knihovnami, jako například AJAX, Prototype atd.

Na rozdíl od většiny programovacích jazyků, jazyk JavaScript nemá žádnou koncepci vstupu nebo výstupu. Je navržen tak, aby jako skriptovací jazyk fungoval v hostitelském

prostředí, a je až na hostiteli poskytnout mechanismy pro komunikaci s vnějším světem. Nejběžnějším hostitelským prostředím je internetový prohlížeč, avšak JavaScriptový interpret lze najít také v Adobe Acrobat, Photoshop, SVG images, Yahoo!'s Widget engine stejně jako na straně serveru jako je například platforma Node.js. JavaScript avšak zahrnuje také NoSQL databáze, jako je například open source Apache CouchDB či MongoDB, nebo desktopové prostředí, například GNOME, jeden z nejpopulárnějších GUI pro operační systémy GNU/Linux. (Richardson, 2015)

JavaScript je objektově orientovaný dynamický jazyk, má datové typy a operátory, objekty a metody. Jeho syntaxe pochází z jazyků Java a C, takže mnoho struktur z těchto jazyků platí stejně pro JavaScript. Jedním z klíčových rozdílů je to, že JavaScript nemá třídy, místo toho, funkcionalita tříd se provádí pomocí objektových prototypů. Dalším hlavním rozdílem je, že funkce jsou objekty, takže je možné v nich vykonat kód a předat je dál jako kterýkoliv objekt.

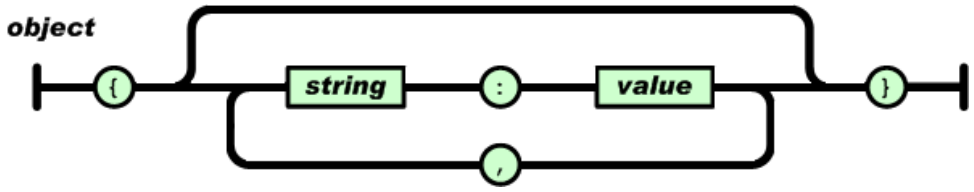
2.5 JSON

JavaScript Object Notation, zkráceně JSON, je odlehčený formát pro výměnu dat a je jednoduše čitelný i zapisovatelný člověkem a snadno analyzovatelný i generovatelný strojově. (Crockford, 2013) JSON se používá především k přenosu dat mezi serverem a webové aplikace, podobně jako jeho alternativa XML.

První, kdo specifikoval a zpopularizoval JSON byl Douglas Crockford. I když byl původně JSON založen na podmnožině skriptovacího jazyka JavaScript, konkrétně na Standard ECMA-262 3. Edition z roku 1999, a běžně se používá u tohoto jazyka, je datový formát jazykově nezávislý. Funkce pro parsování a generování JSON formátu je snadno dostupný pro širokou škálu programovacích jazyků. JSON je založen na dvou strukturách:

- kolekce párů název/hodnota. Ta bývá v rozličných jazycích realizována jako objekt, záznam (record), struktura (struct), slovník (dictionary), hash tabulka, klíčový seznam (keyed list) nebo asociativní pole,
- seřazený seznam hodnot. Ten je ve většině jazyků realizován jako pole, vektor, seznam (list) nebo posloupnost (sequence).

Struktura JSON formátu má pevně danou konstrukci, například na obrázku 2.1 můžeme vidět objekt, což je neuspořádaná množina párů název/hodnota. Objekt je uvozen znakem { a zakončen znakem }. Každý název je následován znakem dvojtečky a páry název/hodnota jsou pak odděleny čárkou.



Obr. 2.1 Konstrukce zápisu JSON objektu (Zdroj: Crockford, 2013)

2.6 Node.js

Node.js je open source multiplatformní běhové prostředí na straně serveru, který je napsán ve skriptovacím jazyce JavaScript a lze ho spustit v operačních systémech OS X, Microsoft Windows, Linux, FreeBSD a IBM i. Tvůrci a vývojáři Node.js ho na oficiální stránkách www.nodejs.org definují jako *„platformu vytvořenou na JavaScriptovém běhovém prostředí Chrome pro snadné vytváření rychlých a škálovatelných webových aplikací. Node.js využívá model událostí a asynchronní I/O operace pro minimalizaci režie procesoru a maximalizaci výkonu a proto je ideální pro datově náročné aplikace běžící v reálném čase v distribuovaných systémech“*.

JavaScript je již dlouhodobě světově nejpopulárnějším programovacím jazykem a díky velkému rozšíření internetu napodobil cíl jazyka Java, a to „napsat jednou, spustit kdekoliv“. (O'Grady, 2015) V roce 2005, kdy se začal hojně využívat AJAX, se z JavaScriptu stal jazyk, který není pouze doplňkem internetových aplikací, ale možné v něm také psát rozsáhlé aplikace. Jeden z prvních, který to potvrdil, byl Google Maps nebo Gmail, avšak v současnosti jej využívají také sociální sítě, jako jsou Twitter či Facebook, nebo GitHub, webová služba podporující vývoj softwaru při používání verzovacího nástroje Git.

Roku 2008, kdy vznikl Google Chrome, se výkon JavaScriptu ještě zvýšil díky konkurenci mezi vývojáři internetových prohlížečů, především pak Mozilla, Microsoft, Apple, Opera a Google. Výkon těchto JavaScriptových virtuálních strojů záleží také na typu webové aplikace. Příkladem může být jslinux, počítačový emulátor běžící na JavaScriptu, kde

je možné načíst Linuxové jádro, komunikovat s terminálem a zkompilovat program v jazyce C, a to všechno v internetovém prohlížeči.

Node využívá V8, virtuální stroj prohlížeče Google Chrome, pro programování na straně serveru. V8 dává Node velký výkon, protože vynechává middleware a preferuje přímou kompilaci do nativního kódu stroje bez nutnosti použití překladače. Díky tomu, že Node využívá JavaScript na straně serveru, tak z něj plynou určité výhody:

- programátoři mohou psát webové aplikace v jednom programovacím jazyce, čímž umožňuje kód sdílet mezi klientem a serverem, například při znovupoužití stejného kódu pro validaci,
- JSON je v současnosti velmi populární formát pro výměnu dat a pro JavaScript je přirozený,
- JavaScript je jazyk využívaný v různých NoSQL databázích, jako jsou CouchDB nebo MongoDB, takže jejich ovládaní je velmi jednoduché,
- Node využívá virtuálního stroje (V8), který dodržuje standardy ECMAScript, podle kterých se řídí i internetové prohlížeče.

Pro mnoho programovacích jazyků je typické využívat konvenčního synchronního „blokujícího“ I/O modelu. Jako příklad můžeme vidět zde:

```
var obsah = fs.readFileSync(,/etc/soubor/‘);
console.log(obsah);
console.log(,Dělej něco jiného‘);
```

Jako první načtene do proměnné obsah soubor, poté jej vypíšeme a nakonec provedeme jinou operaci. Čeho je zde důležité si všimnout, je že se nám druhý výpis ukáže až tehdy, když je načten celý obsah souboru. Mnoha webovým aplikacím tento model vyhovuje a je poměrně jednoduché se jím řídit. Co je ale problém, je že například čtení blokuje celý proces a server nedělá nic jiného, dokud není I/O operace dokončena. Ta může trvat od 10ms do několika minut podle toho, jaká je latence I/O procesu. Latence však může nastat také z různých neočekávaných důvodů:

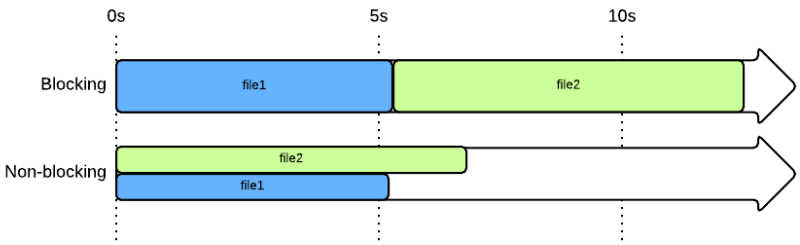
- disk provádí určité udržbové operace, kdy aktuálně pozastavil čtení / zapisování,
- databázový dotaz je pomalejší z důvodu větší databázové zátěže,
- získání zdroje z určité stránky je z nějakého důvodu zpomalené.

Typicky když je server zablokován díky nějaké I/O operaci, je využito více vláken. Běžná implementace je využít jedno vlákno pro jedno připojení a nastavení vláknového poolu pro tyto připojení. Vlákna si můžeme představit jako pracovní místa, ve kterých procesor pracuje na jednom úkolu. V mnoha případech je vlákno zahrnuto uvnitř procesu a udržuje si vlastní pracovní paměť. Každé vlákno pak zpracovává jedno nebo více připojení serveru, takže řízení vláken může být velmi náročné. Taktéž pokud je třeba velký počet vláken manipulujících s více připojeními serveru, můžou se zvýšit nároky na zdroje operačního systému. Vlákna totiž spotřebovávají jak CPU, tak také RAM.

Node.js avšak využívá asynchronního „neblokujícího“ I/O modelu, který si můžeme ukázat na upraveném kódu provádějící stejný proces jako předchozí synchronní model:

```
fs.readFile(,/etc/soubor/‘, function(err, obsah) {
   console.log(obsah);
});
console.log(,Dělej něco jiného‘);
```

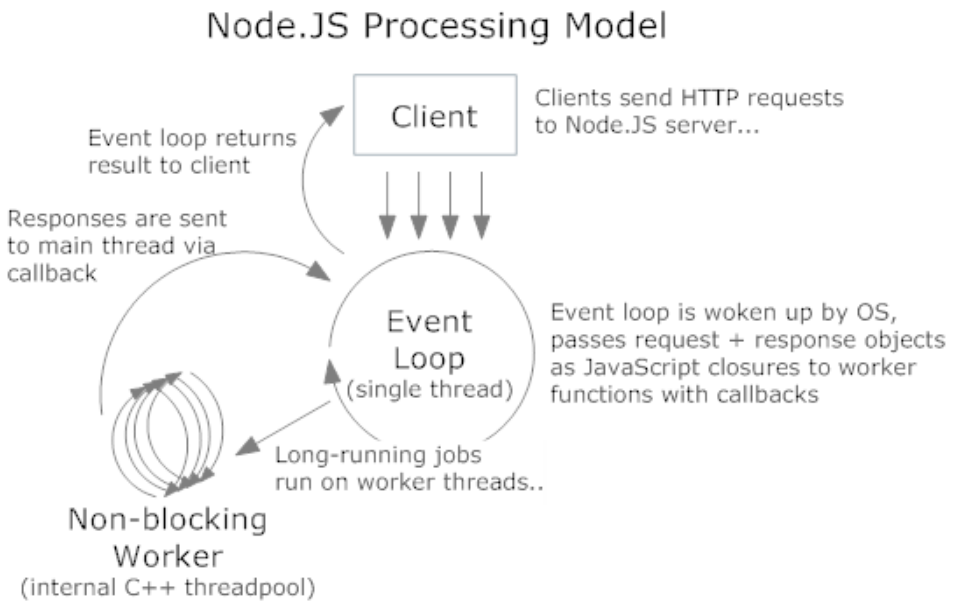
Stejně jako předtím otevřeme soubor a začneme z něj číst. Zde ale nastává změna, protože server nečeká, až se načte celý soubor, ale pokračuje v jeho běhu, tedy vypíše druhou zprávu. Obsah zprávy se vypíše až tehdy, když je načten celý soubor. Využívá se zde tzv. callbacku, tedy že se proces v těle metody vykoná až když je splněna jeho podmínka.



Obr. 2.2 Srovnání rychlosti blokujícího a neblokujícího procesu
(Zdroj: http://openclassrooms.com/courses/ultra-fast-applications-using-node-js/node-js-what-is-it-for-exactly)

Z obrázku 2.2 je pak patrné, že asynchronní model je mnohem rychlejší, zde konkrétně při načítání dvou souborů, kde synchronní model trvá více než deset sekund zatímco v asynchronním modelu jsou soubory (respektive druhý soubor) načten téměř za sedm sekund.

Node.js využívá libuv, což je multiplatformní podpůrná knihovna se zaměřením na asynchronní vstupy / výstupy (I/O). V tomto asynchronním I/O modelu jsou operace otevření, čtení a zapisování zdrojů (například soketů nebo souborů) řízeny systémem „neblokovat volající vlákno“ a vykonat proces jsou-li k dispozici nové data nebo se vyskytla nová událost. Následuje zjištění, ke kterému požadavku událost patří a pokračovat na jeho zpracování.



Obr. 2.3 Proces události v Node.js (Zdroj: http://www.aaronstannard.com/intro-to-nodejs-for-net-developers/)

Celý proces můžeme vidět na obrázku 2.3, kdy klient odešle HTTP požadavek, na ten zareaguje hlavní událostní smyčka, spustí požadavek na jeho vykonání přeposlání požadavku na vláknový pool. C++ vlákna pak právě využívají knihovnu libuv, která zajišťuje asynchronní I/O. Mezitím co je proces vykonáván, tak je možné přijímat další požadavky, protože pool nečeká ani nespí (nelze ho dočasně pozastavit ani uspat). Stejným způsobem se pak zpracovávají SQL dotazy nebo probíhá čtení ze souboru.

2.6.1 Balíčky pomocí Node Package Manager

Node Package Manager, zkráceně npm, je správce balíčků pro JavaScript, především pro Node.js. Tento správce usnadňuje JavaScriptovým vývojářům sdílet kód, který vytvořili za účelem vyřešit určitý problém, jiným vývojářům pro znovupoužití tohoto kódu v jejich aplikacích. V případě, že jsme závislí na tomto kódu od jiných programátorů, npm ulehčuje

kontrolovat jejich aktualizace a následně je stáhnout. Tyto malé části znovupoužitelného kódu se nazývají balíčky nebo také někdy moduly. Balíček se skládá pouze ze složky, jednoho nebo více souborů v ní a také obsahuje soubor s přesně definovaným názvem package.json, který obsahuje metadata o konkrétním balíčku. Typická aplikace, jako je například internetová stránka, závisí na desítkách i stovkách balíčků. Hlavní myšlenkou je vytvořit malé části, které zaručeně řeší určitý problém, a díky tomu zkomponovat větší a komplexnější řešení skládající se z těchto menší částí. (Dayley, 2014)

Všechny balíčky je možné najít na oficiální stránce npm, která obsahuje různé druhy nejen pro Node.js. Je zde možné nalézt mnoho balíčků, které lze jednoduše stáhnout i ovládat pomocí příkazového řádku. Například pokud chceme nainstalovat balíček obsahující dynamický jazyk kaskádových stylů LESS, stačí do příkazového řádku napsat toto:

```
$ npm install less
```

a modul se sám stáhne, nainstaluje a je okamžitě připravený k použití v projektu. Výhodou npm je také to, že nezřídka mívají balíčky závislosti na jiné balíčky. To by znamenalo, že bychom museli zjistit, jaké to závislosti jsou a ručně je pak po jednom instalovat. Díky npm se však není třeba o toto starat, protože společně s balíčkem se i automaticky stáhnou a nainstalují všechny další moduly, které jsou pro jeho běh potřeba. Instalaci je možné rozdělit na dva typy. Tím prvním je globální instalace s globálním parametrem:

```
$ npm install yo -g
```

díky kterému je možné balíček využívat z příkazového řádku kdekoliv. Druhým typem je lokální instalace, která se váže k danému projektu a je možné jej spustit pouze v jeho rámci. Lokální balíčky se pak samy automaticky instalují do složky node_modules, která se nachází v kořenovém adresáři projektu.

2.6.2 Express.js

Express.js je nejpopulárnějším a nejpoužívanějším webovým frameworkem pro Node.js, jeho nejznámější alternativy jsou pak Koa nebo Hapi. Je na něm založeno mnoho webových aplikací, včetně například populárního frameworku Sails.js. Za jeho vznikem stojí známý JavaScriptový vývojář TJ Holowaychuk, který jeho první verzi zveřejnil roku 2009 na

GitHubu, kde byl framework víceméně pouze popsán. O rok později vznikla první oficiální verze 0.0.1. (Glock, 2014)

Express.js je založený na jednom z hlavních Node.js modulu http a na komponentě Connect, která byla původně oddělena, a bylo třeba ji manuálně doinstalovat, avšak později se stala součástí instalace Node.js. Tyto části se hromadně nazývají jako middleware. Samotný middleware obsahuje libovolný počet funkcí, které jsou invokovány směrovací vrstvou Express.js dříve, než dojde ke konečnému cíli. Tyto funkce pak můžeme nazvat jako zásobník middleware, protože jsou vždy invokovány v pořadí v jakém byly přidány. (Shtylman, 2014) Tyto middleware jsou základními kameny filozofie tohoto frameworku, která je „konfigurace před konvencí“. Někteří programátoři jazyka Ruby srovnávají Express.js k frameworku Sinatra, který má velmi odlišný přístup na rozdíl od frameworku Ruby on Rails, který upřednostňuje „konvenci před konfigurací“. Jinými slovy, programátoři si mohou vybrat jakoukoliv knihovnu, kterou potřebují pro konkrétní projekt. Tento přístup jim poskytuje určitou flexibilitu a schopnost si projekt plně přizpůsobit. (Cantelon, 2013)

Express.js řeší mnoho problémů co se týče psaní rutin, tedy není potřeba psát stále dokola ty samé části kódu, které řeší sessions, routy, zachytávání chyb, extrahování URL parametrů atd. Framework poskytuje pro webovou aplikaci také strukturu model-view-controller (MVC). Modelová část pak musí být dodána společně s knihovou pro danou databázi, například Mongoose pro NoSQL databázi MongoDB.

Framework je npm balíček, který je závislý na aplikaci. To znamená, že každý projekt, který je postaven na Express.js, musí mít jeho zdrojové soubory umístěné v lokální složce node_modules v kořenovém adresáři projektu.

2.7 AngularJS

AngularJS je JavaScriptový open source framework pro tvorbu dynamických webových stránek založených na AJAXu. Obecně se dá říct, že složitost při vytváření vysoce škálovatelných a komplexních AJAX aplikací je velká. AngularJS si proto klade za cíl minimalizovat tuto složitost tím, že nabízí kvalitní prostředí pro jeho vývoj a testování.

V současnosti existuje řada softwarových architektur, které oddělují reprezentaci informace od její uživatelské interakce. Hlavní myšlenkou těchto architektur je znovu použitelnost kódu a separace jednotlivých úkolů. Nejpopulárnější a nejpoužívanější softwarovou architekturou je MVC, neboli Model-View-Controller. Aplikační struktura MVC

byla představena již v sedmdesátých letech dvacátého století jako součást čistě objektově programovacího jazyka Smalltalk. Od té doby se architektura MVC stala natolik populární, že byla součástí téměř každého vývojového prostředí, kde bylo použito uživatelské rozhraní. Hlavní myšlenkou MVC je jasně rozdělit aplikaci na řízení s její daty (model), aplikační logiku (controller) a prezentování dat uživateli (view). V praxi to pak funguje tak, že view získá data z modelu, které se zobrazí uživateli. Když uživatel pracuje s aplikací, controller zareaguje změnou dat v modelu a nakonec model upozorní view, že se data změnila, takže může aktualizovat data reprezentována uživateli. (Green, 2013)

Mimo MVC také existuje několik známých softwarových architektur, jako je například MVP, neboli Model-View-Presenter. MVP vychází z velmi podobné architektury jako je MVC a presenter hraje roli prostředníka stejně jako controller v MVC. Nicméně v MVP je veškerá prezentační logika přesunuta do presenteru a vrstva view představuje tu „hloupou“ část, která má za úkol pouze zobrazit data uživateli. Jiným příkladem softwarové architektury může být MVVM, neboli Model-View-View-Model.

Pro aplikaci založené na AngularJS nezáleží, kterou architekturu použijeme, a proto se občas pro její architekturu využívá pojem MVW, neboli Mode-View-Whatever, kde poslední vrstva představuje cokoliv. Základním konceptem MVW je to, že všechny definice jsou spojeny s pojmem modul. Moduly lze pak agregovat do kompletní webové stránky. Moduly mohou být závislé na dalších modulech, takže integrování jednoho modulu do projektu může přinést novou funkcionalitu, na které daný modul závisí. AngularJS poskytuje velkou sadu API definující tyto moduly a propojuje je mezi sebou pomocí dependency injection. (Patel, 2013)

AngularJS nabízí mnoho funkcí, které ulehčují vývoj webových aplikací. Podporuje oboustranné vázání dat, šablonovací systém, jednoduchou REST interakci, vlastní tvorbu komponent, více pohledů, směrování atd. AngularJS také dobře vychází s jinými knihovnami nebo frameworky, například je jej možné kombinovat s jQuery. AngularJS sám o sobě nevyžaduje žádné závislosti, takže je možné využít libovolný počet modulů, případně si projekt škálovat podle sebe.

Framework AngularJS se odvíjí od přesvědčení, že deklarativní programování je lepší než imperativní programování když jde o vývoj uživatelského rozhraní a využívání více komponentů, zatímco imperativní programování je více vhodné pro vyjádření podnikové logiky. Dalšími myšlenkami AngularJS jsou:

- oddělení DOM manipulace od aplikační logiky. Tím se výrazně zlepšuje testovatelnost kódu,
- pohlížení na testování aplikace je stejně důležité jako samotné programování aplikace. Obtížnost testování je velmi ovlivněna tím, jak je kód strukturován,
- oddělení klientské části od serverové části. To umožňuje paralelní vývoj a umožňuje znovu použití kódu na obou stranách,
- framework vede vývojáře skrze celou cestu při vytváření aplikace, od návrhu uživatelského rozhraní přes programování aplikační logiky až k testování,
- udělat běžné triviální a naopak složité úkoly jednoduše řešitelné.

2.8 NoSQL databáze

Již více než desetiletí se používají relační databáze k ukládání strukturovaných dat. Tyto data jsou rozděleny do skupin, známé jako tabulky, ve kterých jsou dobře definované jednotky dat, jako jsou typ, velikost a další omezení. Každá jednotka údajů je známá jako sloupec, zatímco každá jednotka skupiny se nazývá řádek. Sloupce pak mohou mít relace definované mezi sebou, například rodič-dítě. A díky tomu, že konzistence je jedním z klíčových faktorů, je horizontální škálování, tedy že data jsou rozdělena a každý uzel obsahuje pouze část údajů, velmi obtížné, ne-li nemožné. Se vzestupem velkých webových aplikací, se zrodily také ne-relační databáze, obecně označovány jako NoSQL databáze. Jedním z problémů, které NoSQL řeší, jsou právě horizontální škálování. (Vaish, 2013)

NoSQL je termín používaný pro označení uložiště, které nenásleduje tradiční RDBMS model, konkrétně data jsou ne-relační a nevyužívá se dotazovacího jazyku SQL. Tento termín se používá pro databáze, které se snaží řešit problém se škálovatelností a dostupnosti na rozdíl od atomicity a konzistence. NoSQL není databáze ani žádný typ databáze. Ve skutečnosti je to termín používaný k „odfiltrování“ určité sady databází.

Tradiční RDBMS aplikace se zaměřují především na ACID transakce:

- Atomicity - databázová transakce je jako operace dále nedělitelná (atomární). Provede se buď jako celek, nebo se neprovede vůbec,
- Consistency - při a po provedení transakce není porušeno žádné integritní omezení,
- Isolation - operace uvnitř transakce jsou skryty před vnějšími operacemi,
- Durability - změny, které se provedou jako výsledek úspěšných transakcí, jsou skutečně uloženy v databázi a již nemohou být ztraceny.

Jakkoliv se tyto vlastnosti mohou zdát nepostradatelné, tak jsou zcela neslučitelné se škálovatelnou dostupností a výkonem. Jako příklad si to můžeme uvést na jednom z největších e-shopů Amazon, který by tento systém potenciálně praktikoval. Pokud by si uživatel chtěl koupit knihu, transakce by uzamkla část databáze, konkrétně zásoby, a každý další člověk na světě bude muset počkat, dokud daný uživatel nedokončí transakci. Amazon by musel použít kešovaná data, případně odemknout některé záznamy, což by ale vedlo k nekonzistenci. V nejhorším případě, kdy by se jednalo o poslední kus, by musel Amazon pokaždé zasílat omluvný mail, že zboží již není na skladě. Z uvedeného příkladu jde vidět, že v případě potřeby velkého výkonu a vysoké náročnosti není RDBMS tolik vhodný.

Roku 2001 Eric Brewer, profesor na americké vysoké škole University of California v Berkeley a jeden ze zakladatelů Google, přišel s myšlenkou Eventual consistency, která nemá český překlad. Prvky Eventually consistent jsou často klasifikovány jako sémantika BASE na rozdíl od tradičního ACID. BASE se skládá z:

- Basic availability – každý požadavek má garantovanou odpověď – jestli proces dopadl úspěšně nebo ne,
- Soft state - stav systému se může v průběhu času měnit, občas bez jakéhokoli zásahu (pro případnou konzistenci),
- Eventual consistency - databáze může být dočasně nekonzistentní, ale ve výsledku již konzistentní bude.

Eric Brewer také dodal, že pro distribuované systémy není možné zajistit simultánní konzistenci, dostupnost a částečnou toleranci. Tento problém je také znám jako CAP teorém nebo také jako Brewerův teorém. (Voroshilin, 2012)

NoSQL má kromě vysoké škálovatelnosti také řadu dalších výhod, jako jsou například:

- nemají schéma – téměř všechny NoSQL implementace nemají pevně dané schéma. To znamená, že není třeba předem přemýšlet nad definování databázové struktury a je jednoduché ji měnit během vývoje,
- snížená doba vývoje – odpadá totiž problém s řešením složitých SQL dotazů,
- rychlost – přenášení dat je velmi rychlé, což ocení především mobilní uživatelé

Seznam nejpoužívanějších implementací NoSQL databází je možné vidět v tabulce 2.1, která je rozdělena podle typu struktury ukládání dat, které dané databáze používají.

Dokument	Klíč-hodnota	XML	Sloupec	Graf
MongoDB	Redis	BaseX	BigTable	Neo4j
CouchDB	Membase	eXist	Hadoop / HBase	FlockDB
RavenDB	Voldemort		Cassandra	InfiniteGraph
Terrastore	MemcacheDB		SimpleDB	
			Cloudera	

Tab. 2.1 Seznam NoSQL databází podle typu (Zdroj: Vaish, 2013)

2.8.1 MongoDB

MongoDB vznikl ze slova *humongous*, tedy neobyčejně velký, a představuje poměrně nový typ databáze, která nemá žádný koncept tabulek, schémata, SQL jazyka nebo řádků. Také se neřídí transakčními ACID principy, nemá joiny, cizí klíče a další prvky, které bývají pro programátory problémovou záležitostí. MongoDB je tedy velmi odlišná databáze například od tradičního relačního řídícího databázového systému, zkráceně RDBMS.

Hlavní filozofií, o kterou se opírá MongoDB je ta, že jedna velikost není vhodná pro všechno. Po mnoho let byly tradiční SQL databáze uložištěm pro všechny typy. Nezáleželo na tom, zda data jsou vhodná pro ukládání do relačního modelu (který využívaly všechny RDBMS databáze jako například MySQL, PostgresSQL, SQLite, Oracle, MS SQL Server a další), data zde byla stejně uložena. Jedním z důvodů proč tomu tak bylo, je, že je bylo mnohem jednodušší, a také bezpečnější, zapisovat a pak jej číst z databáze, než aby se data zapisovala třeba do souborového systému. (Plugge, 2010)

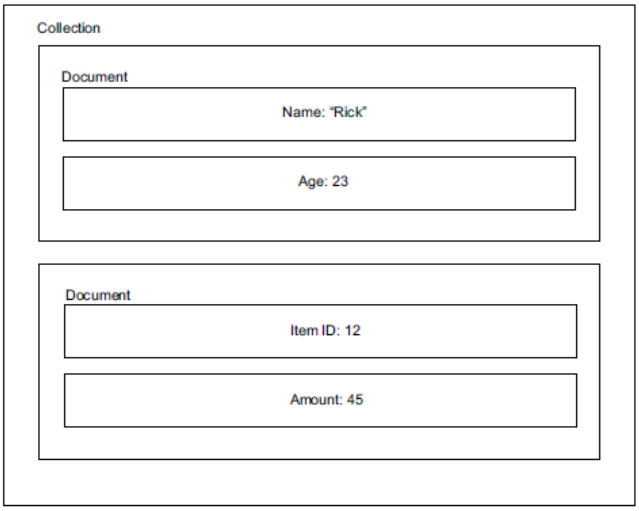
Vývojáři MongoDB se proto rozhodli, že nepůjdou cestou vytvořit další typ databáze, která se snaží řešit všechno pro všechny. Místo toho zamýšleli vytvořit databází, která bude pracovat s dokumenty namísto s řádky, byla velmi rychlá, velmi škálovatelná a jednoduchá pro použití. Aby toho mohli dosáhnout, museli udělat určitá omezení, ze kterých vyplývá, že databáze MongoDB nemusí být vhodná pro všechny typy aplikací. Například nedostatek přítomnosti transakční podpory znamená, že MongoDB není vhodné pro bankovní aplikace. Tento problém se však dá řešit různými hybridními způsoby, jako je například použití RDBMS pro bankovní komponenty a MongoDB pro ukládání dokumentů.

Zvyšování výkonu pro relační databáze je obvykle přímočarou záležitostí – koupí se větší, rychlejší server. Tento způsob avšak funguje jen do té doby, dokud již není k dispozici koupit větší server. V ten okamžik je jedinou možností rozdělit databázi na dva servery a tím se

dostává do problémů většina databází. Například u MySQL ani PostgresSQL není možné rozdělit jednu databázi na dva servery, kde oba mohou číst a zapisovat (často označované jako active/active cluster). (Han, 2012) Ačkoliv například Oracle tento problém dokáže využít díky Real Application Clusters (RAC) architektuře, je třeba počítat s dalšími požadavky, jako je více serverů, sdílená uložiště nebo potřeba několika softwarových licencí.

Problém existence active/active cluster na dvou serverech spočívá při dotazování databáze, protože databáze musí najít všechny relevantní data a dát je dohromady. RDBMS zahrnuje mnoho způsobů, jak zvýšit výkon databází, avšak všechny spoléhají na to, že mají k dispozici kompletní obraz databáze. A zde nastává problém, protože k dispozici je pouze polovina.

Tento problém řeší MongoDB jednoduchým způsobem, kompletně ho obejde. MongoDB ukládá data v BSON dokumentech, takže data jsou „soběstačná“ a každý záznam v MongoDB kolekci může mít úplně jiné schéma, jak je možné vidět na obrázku 2.4. Ačkoliv jsou si podobné dokumenty uložené společně, samotné dokumenty nemají žádné vazby s jinými. Protože dotazy v MongoDB dokumentu hledají specifický klíč a hodnotu, je možné požadavek rozšířit do libovolného množství serverů. Každý server pak kontroluje svůj vlastní obsah a vrací výsledek. Tento efektivní způsob tak dovoluje velkou horizontální škálovatelnost a rychlost. Je však třeba ještě poznamenat, že MongoDB neposkytuje master/master replication, kde oba dva oddělené servery mohou přijímat požadavky na zapisování. MongoDB avšak jakožto NoSQL databáze nabízí sharding. Shard je část databáze, která vznikne, pokud budeme chtít držet určité záznamy zvlášť, místo toho, aby byly všechny pospolu. Každý tento shard může být pak umístěn na odlišném databázovém serveru nebo klidně i geografickém místě. Tato technika má řadu výhod. Poněvadž je databáze rozdělena a distribuována na více serverech, počet záznamů v každé databázi je menší. To zmenšuje velikost indexace a tím i dobu vyhledávání. Dále pokud by vypadl jeden ze shardů (databázových strojů), tak se stále ke zbytku záznamů dostaneme, protože jsou umístěné zvlášť. Navíc můžeme jeden dotaz spustit paralelně na několika shardech a potom pomocí sjednocení získat jeden výsledek. (Tiwari, 2011) Výhodou tohoto způsobu je to, že zatímco některé řešení umožňují mít pouze dvě hlavní databáze, MongoDB může potenciálně škálovat až stovky strojů stejně snadno, jako by běžely pouze na dvou.



Obr. 2.4 Kolekce v MongoDB nemá žádné definované schéma (Zdroj: http://nodeqa.com/nodejs_ref/38)

2.9 REST

REST je nový přístup v systémové architektuře a lehkou variantou k webovým službám. REST značí Representational State Transfer a spoléhá na bezestavovou kešovatelnou komunikaci mezi klientem a server s využitím HTTP protokolu. Jedná se o styl architektury, který je navržen pro navrhování síťově propojených aplikací a hlavní myšlenkou využít tradičního HTTP protokolu pro komunikace mezi stroji, na rozdíl od využívání komplexních mechanismů jako je CORBA, RPC nebo SOAP.

Samotný REST nemá žádný specifický protokol, který by musel využívat, avšak obvykle se pro REST využívá protokol HTTP. Internetové prohlížeče využívají pouze malý zlomek vlastností HTTP. Non-RESTful, tedy ty, které nevyužívají REST, technologie jako jsou SOAP a WS-* využívají HTTP striktně jako přenosový protokol a tím využívají jen malou část jeho možností. Dalo by se říct, že SOAP a WS-* využívají HTTP především k vytváření tunelů skrz firewally. (Burke, 2011)

HTTP je synchronní síťový protokol na bázi požadavek / odpověď a používaný pro distribuované, kolaborativní, dokumentově založené systémy. Jedná se o protokol využívaný především ve světě Internetu a jeho podstata není nijak složitá – klient odešle zprávu požadavku skládající se z použité metody HTTP, zavolá se lokace požadovaného zdroje, nastaví se sada hlaviček a případně se přidá nepovinné tělo zprávy, které může obsahovat mnoho forem, například HTML, čistý text, XML, JSON nebo binární data. Například:

```
GET /aplikace/index.html HTTP/1.1
Host: example.cz
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
```

Zde můžeme vidět odeslání požadavku, když bychom chtěli z prohlížeče navštívit stránku http://example.cz/aplikace/index.html. GET je metoda požadavku na server a /aplikace/index.html je náš požadovaný zdroj. HTTP/1.1 je verze protokolu a Host, User-

Agent, Accept, Accept-Language, Accept-Encoding jsou všechno hlavičky zprávy. Protože si žádáme o informace ze strany serveru, tak zpráva neobsahuje žádné tělo.

Odpověď od serveru je podobná. Obsahuje verzi HTTP protokolu, kód odpovědi, krátká zpráva, která vysvětluje kód odpovědi, různá sada hlaviček a případně tělo zprávy. Například odpověď na GET dotaz může vypadat takto:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet 2.4; JBoss-4.2.2.GA
Content-Type: text/html

<head>
<title>Aplikace</title>
</head>
<body>
<h1>REST aplikace</h1>
```

Kde kód odpovědi je 200 a status zprávy je „OK“. Tento kód znamená, že žádost byla úspěšně zpracována a že klient obdrží požadované informace. HTTP obsahuje mnoho dalších kódů odpovědí, kde každý má svůj důvod. Zpráva odpovědi obsahuje také tělo, ve které je HTML kód, a proto je hlavička Content-Type zahrnující HTML.

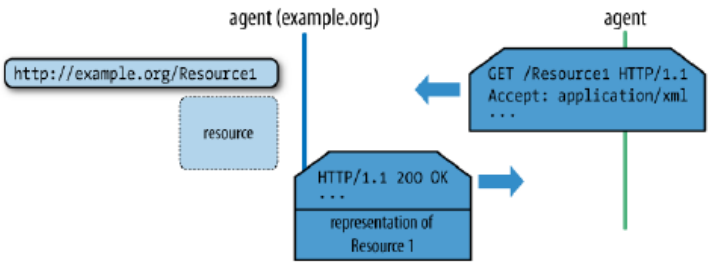
První, kdo popsal REST, byl Roy Fielding, jeden ze spoluautorů protokolu HTTP, ve své dizertační práci „*Architectural Styles and the Design of Network-based Software Architectures*“, kterou napsal již v roce 2000 a proto nepřekvapí, že REST má s HTTP hodně společného. Programátoři si postupně poté začali uvědomovat, že mohou využít koncept RESTu pro distribuované služby a pro tvorbu servisně orientovaných architektur, zkráceně SOA. Myšlenka SOA je ta, že vývojáři navrhují své systémy jako sadu znovu použitelných, oddělených a distribuovaných služeb. Vzhledem k tomu, že tyto služby jsou využívány v síti,

tak je snazší sestavit větší a složitější systémy. Vývojáři proto v minulosti využívali technologie jako DCE, CORBA nebo Java RMI. V dnešní době se v SOA využívají především webové služby založené na SOAP. I když má REST mnoho společného s tradiční SOA aplikací, je zde stále několik zásadní prvků, kterým se liší, například REST je orientován datově, nikoli procedurálně. Webové služby definují vzdálené procedury a protokol pro jejich volání, REST určuje, jak se přistupuje k datům.

Jedním ze základním prvků REST je adresovatelnost, tedy že každý objekt a zdroj v systému je dostupný pomocí unikátního identifikátoru. V REST architektuře je adresovatelnost řešena pomocí URI. Když vytváříme požadavek na informaci v internetovém prohlížeči, píšeme URI, jinak řečeno, každý HTTP požadavek musí obsahovat URI požadovaného objektu. Formát URI má standardizovaný tvar:

```
schéma://host:port/cesta?dotaz#fragment
```

Schéma je protokol, který využíváme pro komunikaci. Pro RESTful webové služby je to obvykle http nebo https. Host je pak DNS název nebo IP adresa, kde poté následuje číslo portu. Host a port reprezentují umístění zdroje v síti. Následuje cesta k danému objektu zahrnující nepovinný dotaz a fragment. Zdroje, identifikátory a akce jsou všechno, co potřebuje pro práci se zdroji umístěných v síti. Například obrázek 2.5 zobrazuje, jak XML reprezentace může být v pořádku vyžádána a doručena pomocí HTTP společně s kódem odpovědi.



Obr. 2.5 Využití HTTP k získání reprezentace zdroje (Zdroj: Weber, 2010)

Dalším principem REST je omezení rozhraní, tedy že existuje určitý konečný počet operací aplikovaného protokolu, které je možné použít. To znamená, že v URI neexistuje žádný „action“ parametr a je možné využít pouze HTTP metody. HTTP má malou, omezenou sadu metod, kde každá metoda má svůj vlastní účel a smysl, jak je možné vidět v tabulce 2.2.

Http metoda	Kolekce dat	Jedna datová položka
GET	Seznam URI identifikátorů jednotlivých položek.	Konkrétní položka v definovaném formátu.
PUT	Nahrazení celé kolekce jinou kolekcí.	Editace konkrétní položky. Pokud neexistuje, bude vytvořena.
POST	Vytvoření nové položky v kolekci a vrácení její URI.	Obvykle nepoužíváno.
DELETE	Odstranění celé kolekce.	Odstranění konkrétní položky.

Tab. 2.2 Data vrácená po provedení jednotlivých akcí na zdroje

I když se může toto omezení zdát na první pohled jako nevýhodné, obsahuje řadu výhod:

- znalost – pokud známe URI určité služby, víme také přesně, jaké existují metody pro daný zdroje. Nepotřebujeme tak žádné IDL soubory popisující, jaké metody jsou k dispozici. Vše co je potřeba, je uživatelská HTTP knihovna,
- interoperabilita – většina programovacích jazyků obsahuje klientskou HTTP knihovnu, takže je velká pravděpodobnost, že lidé, kteří budou chtít naši službu využívat, nebudou potřebovat další věci pro její chod. Například s CORBA nebo s WS-* je třeba nainstalovat jejich specifickou uživatelskou knihovnu stejně jako číst IDL soubory nebo využívat kódu, který vygeneroval WSDL,
- škálovatelnost – protože REST omezení nás nutí využívat předem definované metody, můžeme předpokládat, jak se systém zachová. Toho můžeme využít například při kešování GET požadavku, který je idempotentní a zároveň bezpečný, a tím výrazně snížit síťový provoz.

2.10 Single-page aplikace

Single-page aplikace, zkráceně SPA, je plnohodnotná webová aplikace obsahující pouze jednu stránku, která je výchozím bodem pro ostatní webové stránky a využívá JavaScript, HTML5 a CSS pro všechny front-end interakce. V SPA aplikacích není žádné obnovení či načítání celé stránky (kromě prvotního načtení) a nejsou zde vloženy žádné objekty. Na

druhou stranu se využívá HTML elementu, ve kterém se jeho obsah nahrazuje různými pohledy prostřednictvím front-end směrováním a šablonovacím mechanismem.

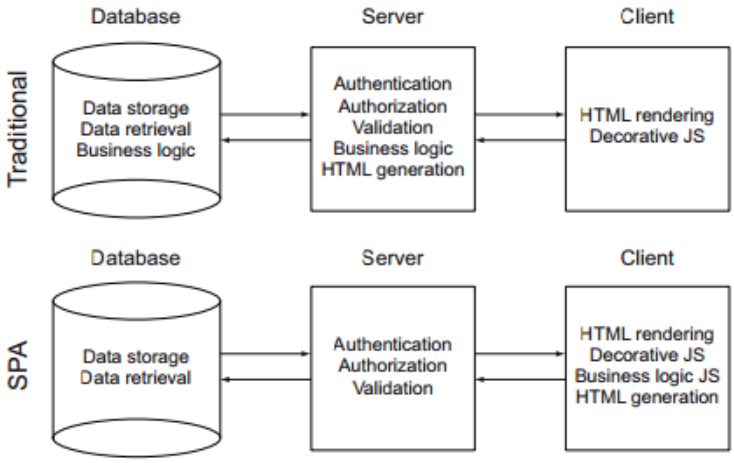
Flash a Java applety se postupně vyvíjely od roku 2000. Java se používala pro komplexní aplikace a dokonce nabízeli kompletní kancelářské balíčky dostupné přes prohlížeč. Flash se stal platformou pro hraní her v prohlížeči a později pro přehrávání videa. Na druhou stranu, JavaScript byl využíván akorát jako kalkulačka, validace formulářů, různé jednoduché efekty a vyskakovací okna. Problém byl takový, že se na JavaScript nedalo spoléhat díky prohlížečům, které umožňovaly jednoduše JavaScript zakázat. (Mikowski, 2013) Přesto však JavaScriptová SPA nabízela řadu výhod oproti Flashi nebo Javy:

- nebyla třeba pluginu – uživatelé měli přístup k aplikaci bez nutnosti instalace pluginu, jeho údržby a dbát na kompatibilitu s operačním systémem,
- menší „nafouknutí“ – SPA využívající JavaScript a HTML obsahovala méně zdrojů, než pluginu, který ke svému provozu potřebuje běhové prostředí,
- jeden klientský jazyk – weboví architekti a programátoři musí znát mnoho programovacích jazyků a datových formátů, jako například HTML, CSS, JSON, XML, JavaScript, SQL, PHP / Java / Ruby / Perl / Python atd. Použití jednoho programovacího jazyka pro všechno na klientské straně snižuje nutnou komplexitu.

Jakmile se JavaScript zdokonalil, mnoho z jeho slabin bylo odstraněno nebo velmi zredukováno a jeho výhod přibylo, například:

- internetový prohlížeč je světově nejpoužívanější aplikací – přístup k JavaScriptové aplikaci již není problém,
- JavaScript v prohlížeči je jedním z největších distribuovaných prostředí – JavaScriptová implementace se rozšířila od desktopových zařízení i na mobily a tablety,
- JavaScript je užitečný pro multi-platformní vývoj – je možné vytvořit SPA aplikaci běžící stejně na Windows, Mac OS X nebo na Linuxu,
- JavaScript se stal velmi rychlým – jeho rychlost se zvyšuje díky konkurenci hlavních internetových prohlížečů, jako jsou Mozilla Firefox, Google Chrome, Opera, Safari a Internet Explorer od Microsoftu.

SPA může využít řadu serverových technologií. Vzhledem k tomu, že se webové aplikace přesunují na stranu klienta, tak se snižují požadavky na server. Obrázek 2.6 ilustruje jak se business logika a HTML vykreslování přesouvá od strany serveru ke klientovi.



Obr. 2.6 Odpovědnosti databáze, serveru a klienta mezi tradičním přístupem a SPA (Zdroj: Mikowski, 2013)

Pokud bychom srovnali tradiční webovou aplikaci a životní cyklus její stránky, tak hlavním rozdílem je povaha požadavku a odpovědi, které se řídí výchozím HTTP požadavkem. V SPA využíváme AJAX k vyžádání dat a výsledná data získáme ve formátu JSON nebo HTML. Jakmile obdržíme potřebná data pro klienta, tak vykreslí jen ta část HTML, která se změnila. Také pohyb mezi stránky se v SPA děje na straně klienta, čímž se odlišuje od tradiční aplikace.

Dalším rozdílem mezi SPA a tradiční webovou aplikací je řízení stavu. Díky tomu, že v SPA uživatel neopouští ani neobnovuje hlavní stránku, můžeme zachovat aktuální stav v paměti prohlížeče. Také pokud bychom chtěli zachovat stav, když uživatel zavře prohlížeč, můžeme využít HTML5 storage. Když si poté znovu otevře prohlížeč, je možné uživatele vrátit tam, kde skončil bez jakékoliv účasti serveru. (Fink, 2014)

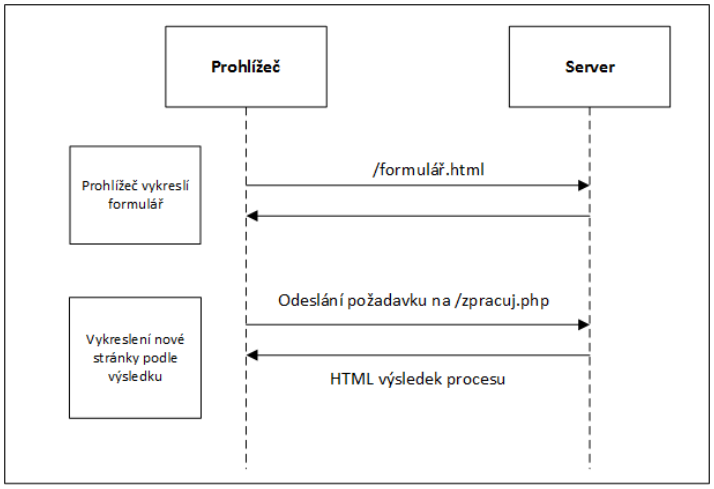
2.11 Real-time aplikace

Jsme zvyklí, že když na internetové stránce klikneme na tlačítko nebo na odkaz, něco napíšeme do textového pole a odešleme, tak to způsobí určitou změnu stránek. Avšak pokud bychom nechali stránku bez dotyku a najednou dostali upozornění, že jsme obdrželi zprávu bez jakékoliv činnosti ze strany uživatele, pak můžeme ty stránky nazvat jako real-time aplikací, tedy aplikací běžící v reálném čase.

Real-time aplikace není žádná novinka, první pokusy vytvořit webovou real-time stránku vznikaly již s použitím Java appletů. Poté následovaly pluginy Flash a ActiveX, které nebyly zaměřeny pouze pro domácí uživatele, ale také se využívali v podnikatelské sféře. V dnešní

době se tento typ webových aplikací rozšířil hlavně díky způsobu, jak je real-time funkcionalita implementovaná a rapidní snížení nákladů. Již není třeba využívat různých triků nebo technických pomůcek, naopak se real-time aplikace stávají standardem ve formě WebSockets a Server-Sent Events (SSE). (Rai, 2013)

Web, a tím i webové aplikace, je postaven na protokolu HTTP. Jedná se o systém založený na požadavku / odpovědi, kde klient posílá požadavek na získání informace serveru a server naopak odpovídá s požadovanými informacemi, nejčastěji ve formátu HTML, XML nebo JSON, které poté vykreslí internetový prohlížeč. Tuto interakci mezi prohlížečem a serverem můžeme vidět na obrázku 2.7.

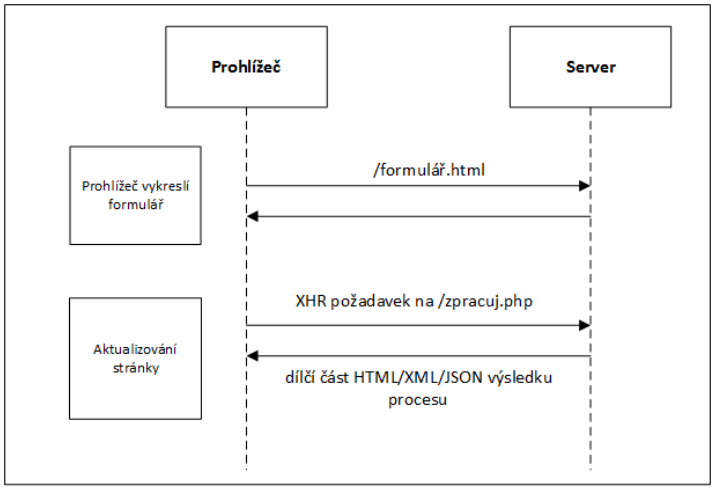


Obr. 2.7 Interakce mezi prohlížečem a serverem (Zdroj: vlastní)

V roce 1995 Sun Microsystems a Netscape oznámili partnerství s tím, že Netscape zahrne do svého prohlížeče Java runtime od společnosti Sun a tím započaly vysoce interaktivní internetové stránky. Ačkoliv si od té doby vysloužily špatnou pověst, applety byly pionýry na poli real-time aplikací a v jejich raných dobách byly používány téměř všude, pro chat, hry nebo pro reklamní banery. Ten samý rok přišel Netscape se skriptovacím jazykem JavaScript a společnost FutureWave Software začala pracovat na animačním softwaru s názvem FutureSplash Animator. Později právě tyto dva produkty zapříčinily postupnou redukci Java appletů z webových stránek. Roku 1996 FutureWave odkoupila firma Macromedia a přejmenovala produkt FutureSplash Animator na Flash, který se stal po další desetiletí nejužívanější platformou pro tvorbu animací, her a dalších interaktivních elementů.

Roku 1999 začal Microsoft používat na své domovské stránce technologii vložených rámů, anglicky iframe, k aktualizaci novinek. Ten samý rok také zveřejnil proprietární

ActiveX rozšíření pro Internet Explorer, pojmenovaným jako XMLHTTP. Nastala éra, kdy se pro všechno začal používat formát XML. Tato XMLHTTP komponenta byla původně zamýšlena k asynchronnímu načítání XML dat ve stránce s využitím JavaScriptu. Brzy jej adoptovaly také prohlížeče Mozilla, Safari a Opera jako XMLHttpRequest, zkráceně XHR. Byl to však Gmail od Google, kdo jako první dokázal efektivně využít technologie XML a JavaScript a to díky AJAXu, který představil Jesse James Garrett. Na obrázku 2.8 můžeme vidět, jak probíhá požadavek na server díky technologii AJAX. Gmail tak ukázal výhody aktualizací běžící v reálném čase a otevřel dveře dalším způsobům využití AJAXu pro získání dat ze serveru, nebo aspoň iluzi, že se tak děje.



Obr. 2.8 AJAX požadavek (Zdroj: vlastní)

Tomuto modelu webových aplikací, kdy se delší dobu drží HTTP požadavek umožňující serveru předat data prohlížeči, aniž by si to o to prohlížeč explicitně řekl, se také říká Comet, termín který definoval Alex Russell na svém blogu v roce 2006. Comet není jediný přístup, naopak definoval několik mechanismů, jak získat data ze serveru klientovi, jako jsou XHR polling, XHR long polling a JSONP long polling. (Rusell, 2006)

Nicméně jeden z problémů long pollingu je ten, když je zde požadavek na obousměrnou komunikaci mezi klientem a serverem. Jakmile je totiž otevřeno HTTP připojení pomocí long pollingu, jediný způsob jak může klient komunikovat se serverem je ten, že se vytvoří nový HTTP požadavek. To znamená, že se zdvojnásobí zdroje, které se využívají. Jednou pro zprávy mezi ze serveru klientovi a poté zprávy od klienta serveru. Velikost důsledku pak záleží na tom, jak velká obousměrná komunikace probíhá, čím více si klient a server vyměňují data, tím jsou více zatíženy zdroje. Další problém tohoto přístupu se nachází mezi

long polling požadavcích, kdy existuje krátký časový úsek, ve kterém je možné, aby data u klienta nebyly synchronizované s daty na serveru. Pouze v okamžiku, když připojení obnovené může klient zkontrolovat, jestli se data nějak změnily. Negativní dopad záleží především na datech, nicméně v aplikacích, kde jsou data velmi citlivé na aktuálnost, to není vhodné řešení.

Potřeba používat několik připojení pro obousměrnou komunikaci je však také omezena internetovými prohlížeči, především těmi staršími. Schopnost JavaScriptu běžet na webové stránce a vytvářet požadavky na server byla dlouho omezena, umožňující posílat požadavky pouze na stejné doméně. (Zalewski, 2009) Například pokud byla stránka www.example.cz/index.html, JavaScript mohl vytvořit požadavek pouze na zdroj nacházející se na www.example.cz nebo na jeho subdoméně. Toto omezení internetových prohlížečů vzniklo z bezpečnostních důvodů, avšak zablokovalo to posílat požadavky na jiné domény. Potřeba tohle vyřešit dala vzniknout mechanismu umožňující sdílení zdrojů (např. fontů, JavaScriptového kódu, atd.) webové stránky pro aplikaci na jiné doméně s názvem CORS z anglického Cross-origin resource sharing. CORS má dobrou podporu v internetových prohlížečích, avšak u některých starších verzích se vykytují stále problémy. Právě u starších prohlížečů to znamenalo, že uživatel mohl mít otevřenou pouze jednu stránku webové aplikace, která využívala long polling nebo streamování. Pokud chtěl otevřít druhou stránku, tak se mu připojení přerušilo. I když toto omezení existuje i v moderních prohlížečích, respektive v jejich novějších verzích, počet možných připojení je již větší.

Předchozí přístup jsou avšak stále jen různé modifikace, aby se dalo využít HTTP a XHR pro obousměrnou komunikaci, které však pro to nebyly uzpůsobeny. S rapidní evolucí internetových prohlížečů, především Firefoxu a Chrome, se začalo také rozšiřovat a využívat HTML5. A právě tato technologie obsahuje dvě metody pro získání dat ze serveru ke klientovi. Jeden je Server-Sent Events, zkráceně SSE, a druhý jsou plně duplexní WebSockets.

Server-Sent Events se snaží standardizovat komunikaci mezi prohlížeči pomocí modelu Comet. V tomto přístupu se využívá JavaScript API k vytvoření zdroje události, to znamená streamu, přes který může server posílat události. Stále se zde však využívá starší XHR a tento přístup je vhodný do aplikací, kde není potřeba plně duplexní komunikace, a jen získáváme aktualizace ze serveru klientovi.

2.11.1 WebSockets

WebSockets nabízejí standardizovanou cestu toho, pro co bylo po dlouho dobu nutné využívat modifikovaného modelu Comet. Cílem bylo dosáhnout obousměrné komunikace mezi serverem a klientem s využitím jednoho připojení. HTML5, které zahrnuje i WebSockets, zajišťuje také podporu komunikace mezi doménami. Pracovní skupina WHATWG definuje WebSocket (2010) jako „*protokol umožňuje dvousměrnou komunikaci mezi uživatelským agentem spouštějící nedůvěryhodný kód provozovaném v kontrolovaném prostředí a mezi vzdáleným hostem, který se rozhodl komunikovat s tímto kódem. Jako model bezpečnosti se využívá Same-origin policy, který je běžně používán internetovými prohlížeči. Protokol se skládá z prvotního handshake a následující jednoduchou zprávou, přenesou pomocí TCP. Cílem této technologie je poskytnout mechanismus pro webové aplikace, které potřebují dvousměrnou komunikaci se serverem a nespoléhat na vícenásobné HTTP připojení (například XMLHttpRequest, iframe nebo long polling).*“

Jednou z hlavním výhod implementací WebSockets je podpora ve škálování. Protože WebSockets využívají jediného TCP připojení pro komunikaci mezi serverem a klientem namísto vícenásobných, separovaných HTTP požadavků, je režie výrazně snížená.

Protože plně duplexová komunikace nemůže být dosažená jen pomocí HTTP, WebSocket definuje zcela nový protokol umožňující připojení klienta k serveru. Toho je dosaženo pomocí otevření HTTP požadavku a poté server zažádá o „vylepšení“ připojení k WebSocket protokolu posláním hlavičky:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

A pokud je požadavek úspěšný, tak server vrátí hlavičku ve tvaru:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Tato výměna se nazývá handshake, česky potřesení rukou, a je vyžadován pro vytvoření WebSocket připojení. Jakmile vznikne úspěšný handshake mezi server a klientem, je vytvořen dvousměrný kanál a obě strany si sobě mohou nezávisle posílat data. Data posílaná po hanshake jsou uložena v rámcích, což jsou důležité části informací. Každý rámec začíná s 0x00 bytem a končí s 0xFF bytem, což znamená, že každá odeslaná zpráva obsahuje kromě svého obsahu pouze dva byty navíc. (Lengstorf, 2013)

2.11.2 Socket.IO

Socket.IO, kterou vytvořil Guillermo Rauch, CTO společnosti LearnBoost, je JavaScriptová knihovna zajišťující real-time, obousměrnou komunikaci mezi klientem a serverem. Skládá se z klientské knihovny běžící v prohlížeči a ze serverové knihovny pro Node.js, kde obě mají téměř identické API. Díky tomu, že se jedná o komponentu Node.js, lze knihovnu jednoduše nainstalovat z příkazového řádku jako npm balíček.

Socket.IO primárně využívá WebSocket protokol a poskytuje stejné rozhraní a ačkoliv se tato knihovna může využívat pouze jako obálka WebSockets, tak se využívá především proto, že poskytuje mnoho funkcí, například broadcasting do více socketů, ukládání dat spojená s každým klientem nebo asynchronní I/O. Další výhodou je to, že funguje i ve starších prohlížečích – až do Internet Explorer 6, zatímco WebSockets fungují až od Internet Explorer 10. (Rai, 2013)

Nevýhodou naopak je to, že Socket.IO neumožňuje využít jiný real-time protokol než je WebSocket. Takže klient, který implementuje Socket.IO nemůže komunikovat například se serverem využívající Long Polling Comet. Právě proto Socket.IO vyžaduje používání jeho knihovny na straně klienta i na straně serveru.

3. Analýza a návrh řešení

Analýza a návrh řešení se považuje za jednu z nejdůležitějších částí projektu. Tuto etapu bychom určitě neměli nijak podceňovat, protože od ní se bude vyvíjet všechno ostatní. Protože se jedná o internetovou aplikaci na zakázku, tak je potřeba nejdříve vykonat analýzu požadavků klienta, která nám určí vlastnosti celého informačního systému. Požadavky můžeme rozdělit na dva základní druhy – funkční a nefunkční. Nicméně ve skutečnosti rozdíly mezi těmito typy požadavků nejsou tak jednoznačné, jak naznačují jejich definice. Uživatelský požadavek týkající se bezpečnosti, například omezující přístup neoprávněných uživatelů, se může jevit jako nefunkční požadavek. Avšak jak je postupně hlouběji analyzován, může tento požadavek vytvářet další požadavky, které jsou jednoznačně funkční, jako například potřeba zahrnout uživatelskou autentizaci pro přístup do systému. To ukazuje, že požadavky nejsou nezávislé a že jeden požadavek často vytváří nebo omezuje jiné požadavky. (Sommerville, 2011)

3.1 Funkční požadavky

Funkční požadavky systému popisují co má systém dělat, jak by měl systém reagovat na konkrétní vstupy a jak by se měl systém chovat v konkrétních situacích. V některých případech můžou funkční požadavky také explicitně popisovat, co by systém dělat neměl.

Tyto požadavky jsou závislé na charakteru vyvíjeného softwaru, resp. aplikace, na jeho uživatelích a také na přístupu vzniku požadavků. Funkční požadavky jsou obvykle popisovány v abstraktní formě, kterým mohou snadno porozumět i uživatele systému. V zásadě specifikace funkčních požadavků systému by měla být jak úplná, tak i konzistentní. Úplnost znamená, že všechny služby vyžadované uživatelem by měly být definovány. Konzistence znamená, že požadavky by neměly mít protichůdné definice. V praxi je však ve velkých a složitých systémech prakticky nemožné dosáhnout konzistentních a úplných specifikací. (Sommerville, 2011) Jedním z důvodů je to, že je snadné udělat chybu a opomenout při psaní určitý požadavek, především pro komplexní systémy. Dalším důvodem je to, že ve velkém systému existuje i mnoho zúčastněných stran. Zúčastněná strana je pak osoba nebo role, která je ovlivněna nějakým způsobem systémem. Zúčastněné strany mají různé, a často nekonzistentní, požadavky. Tyto nesrovnalosti nemusí být zřejmé na první

pohled, problémy se pak mohou objevit až po hlubší analýze nebo až poté, co byl systém dodán zákazníkovi.

Pro vývoj internetové aplikace byly v rámci specifikace uvedeny tyto funkční požadavky:

- Uživatel si bude moct prohlížet veškeré inzeráty, vkládat a upravovat svůj inzerát. Každý uživatel pak bude moct vytvořit pouze jeden inzerát.
- Uživatel se bude moct registrovat, přihlašovat a odhlašovat. Pro přihlášení bude moct využít jak pomocí lokálního účtu, tak také pomocí známých sociálních sítí, jako je Facebook, Google nebo Twitter.
- Přihlášený uživatel si může upravovat svůj vlastní profil kromě své přezdívky. I nepřihlášení uživatelé si budou moci prohlídnout uživatelův profil.
- Aplikace bude obsahovat menu s několika vybranými sporty a také se seznamem všechny regionů České republiky. Tyto sporty a regiony budou obsahovat inzeráty, které pod ně spadají.
- Jeden inzerát musí spadat pouze do jednoho regionu, avšak může obsahovat libovolný počet sportů.
- Jakmile bude přidán nový inzerát, tak se to uživateli objeví v upozornění nacházející se v menu.
- Uživatel si bude moct přidávat a odebírat jiné uživatele jako své přátelé a po jejich přihlášení uvidí, že jsou online anebo jsou naopak offline bez nutnosti obnovovat stránku.
- Uživatel může jinému uživateli poslat zprávu, kdy příjemce tuto zprávu uvidí hned bez nutnosti obnovovat stránku.
- Aplikace bude obsahovat jednoduchou administraci, ke které je umožněn přístup autorizovaným uživatelům na základě jejich přezdívky. V administraci bude moct provozovatel měnit a mazat inzeráty a také přidávat nové sporty.
- Editace klientské ani administrační části nebude požadovat znalost žádných programovacích jazyků a bude prováděna jen pomocí internetového prohlížeče.
- Aplikace bude obsahovat určité bezpečnostní mechanismy.

3.2 Nefunkční požadavky

Nefunkční požadavky jsou omezení týkající se služeb nebo funkcí, které jsou v systému. Patří mezi ně časové omezení, omezení při vývojovém procesu a omezení stanovených standardů. Nefunkční požadavky se často týkají systému jako celku, spíše než jednotlivé funkce systému nebo služeb.

Tento typ požadavků je nezbytným atributem v systému, který určuje, jak funkce mají být provedeny a často odkazují na vlastnosti systému, jako je například spolehlivost, znovu použitelnost, přenositelnost, udržovatelnost, kompatibilita, ověřitelnost, předvídatelnost, bezpečnost, zabezpečení informací, účinné využívání zdrojů, úplnost nebo lidský faktor. (Young, 2001)

Pro vývoj internetové aplikace byly v rámci specifikace uvedeny tyto nefunkční požadavky:

- Aplikaci bude uživatel moct používat ve všech běžně dostupných internetových prohlížečích. Vzhled se bude lišit jen v drobných odchylkách a u starších prohlížečů (např. Internet Explorer 6.0 a nižší) je nejdůležitějších zachování funkcionality.
- Aplikaci bude také moct použít na mobilních zařízeních, tedy její vzhled bude responzivní a přizpůsobí se danému zařízení.
- Aplikace bude využívat JavaScriptovou platformu Node.js společně s klientským frameworkem AngularJS a frameworkem Express na straně serveru.
- Pro ukládání dat se bude využívat NoSQL databáze MongoDB.

3.3 Strukturovaná analýza

Strukturovaná analýza je technika softwarového inženýrství s cílem konstruovat a zdokonalit systémovou specifikaci z daných požadavků. Funkcionální dekompozice využívajíce diagramů datových toků ve strukturované analýze se často využívá v reaktivních systémech (systémy, které reagují na externí události) a mohou poskytnout softwarovou strukturu s menším sémantickým rozdílem programovacího jazyka. (Nakanishi, 2009)

Syntéza je kompozice, analýza je dekompozice. Strukturovaná analýza je tedy strukturovaná dekompozice k dosažení určitého cíle. Obrázky, slova či výrazy pak mohou být začleněny v určité části struktury. (Ross, 1977)

V odpovědi na otázku „proč modelujeme?“ Booch (1999) navrhuje základní důvod – protože můžeme lépe pochopit systém, který vyvíjíme. Wilson (1990) ve své knize rozšiřuje porozumění modelu tím, že definuje čtyři role konceptuálního modelování: vyjasnění si prostoru našeho zájmu, ilustrace koncepce, pomoc při definování struktury a logiky a předpoklad pro vytvoření návrhu. Ačkoliv bychom mohli vytvářet modely pro lepší porozumění aktuálních podnikových procesů, můžeme také vytvářet modely reprezentující různé konceptualizace do budoucnosti, což se může stát například v případě, kdy provádíme výrazné změny aplikace zahrnující redesign podnikových procesů. Z pohledu vývoje systému Booch (1999) tvrdí, že nám modely nejen pomáhají vizualizovat systém jaký aktuálně je nebo jaký bychom ho chtěli mít, ale také nám dovoluje specifikovat strukturu a chování systému a poskytuje šablonu, která nás vede při konstrukci softwarového systému.

Booch (1999) dále argumentuje, že „*model je zjednodušení reality*“ a že „*nejlepší modely jsou spojeny s realitou*“. Tyto definice však můžou vyvolávat otázku, co představuje realitu a jakého tvaru může nabývat jeho spojení. Obecněji řečeno, Wilson (1990) definuje model jako „*něčí explicitní interpretaci pochopení situace nebo něčí představy o situaci. Může být normativní nebo ilustrativní, ale především musí být užitečný.*“ Modely však neodrážejí pouze současnou realitu, ale také se podílejí na konstrukci nových skutečností, respektive co bude. Avšak musíme si být také vědomi omezení modelů a jejich anotací. V modelování současné a potenciální budoucí situace musíme využít našich předpokladů, vyznačit hranice a akceptovat to, že existují určité osobní, kulturní či politické aspekty, které nelze vyjádřit diagramy.

3.3.1 Diagram užití

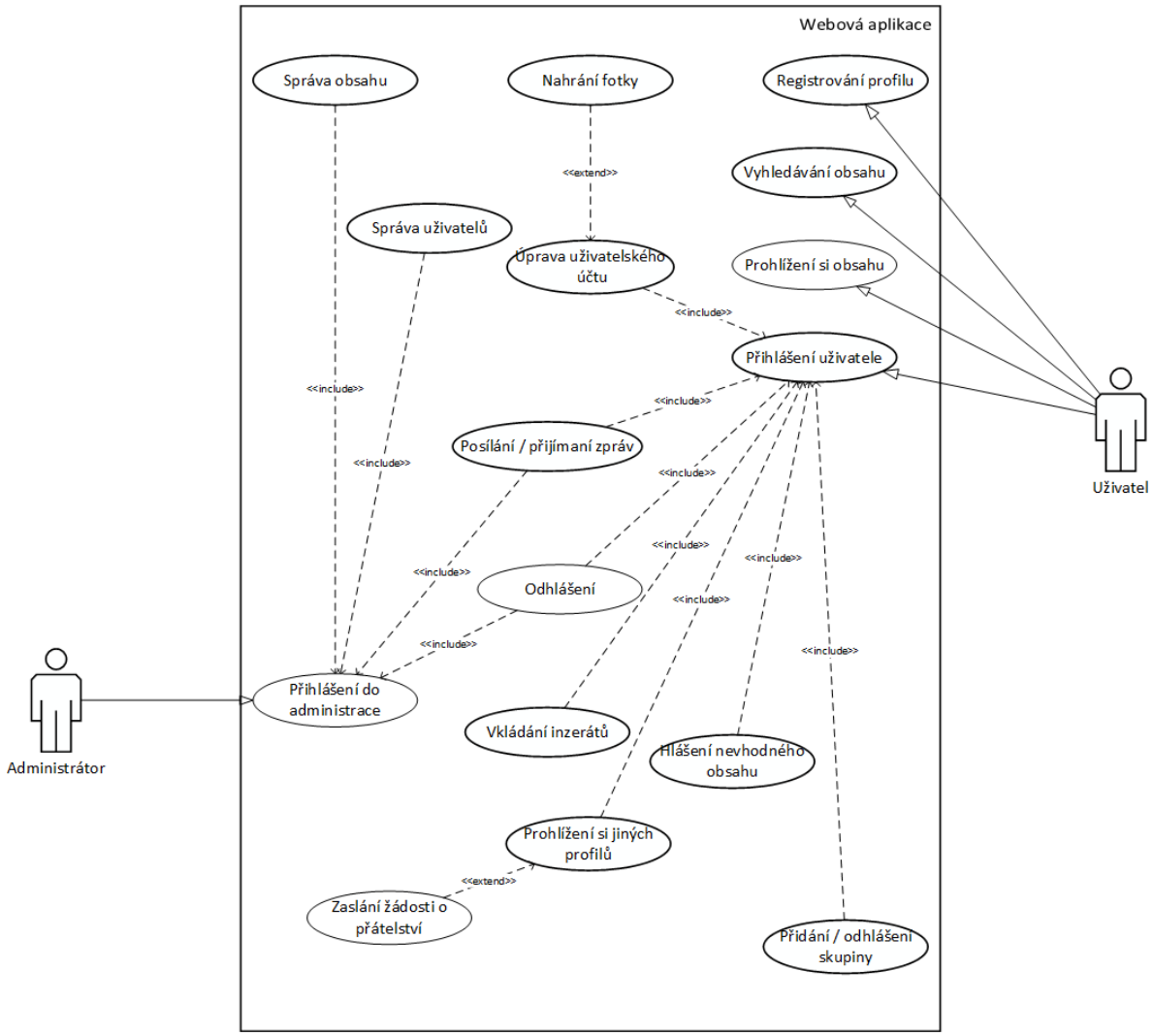
V osmdesátých letech dvacátého století rostla popularita objektově orientovaného programování a společně s ním se pomalu rozvíjely i oblasti návrhu a analýzy systémů. V devadesátých letech pak vznikla řada metodik, které měly své silné i slabé stránky. Metodika Boosch měla silné stránky především při návrhu a v real-time aplikacích, technika OMT od Jima Rumbaugha se zaměřovala na analýzu a datově náročné aplikace a metodologie OMSE Ivara Jacobsona byla zaměřena na modelování podnikových procesů. V roce 1994 se Boosh, Rumbaugh a Jacobson dali dohromady a spojily své nápady k vytvoření sjednoceného

jazyka UML, který obsahuje ty nejlepší vlastnosti z každé metodiky, a vytvořili standard, který v současné době definuje standardizační skupina Object Management Group.

Výchozím bodem pro modelování podnikových požadavků s využitím UML jsou diagramy užití (anglicky use case diagram), což jsou formalizované notace pro modelování systému z hlediska uživatele. Důraz je spíše kladen na to, co systém dělá (jeho chování), než jak toho dosahuje. Diagram užití je tedy specifikace sekvencí akcí, včetně variantních sekvencí, v nich systém, subsystém nebo třída může provádět interakci s externími subjekty. (Suyono, 2006)

Diagram užití typicky reprezentuje některé funkčnosti navrhovaného systému, jak jej vnímá uživatel. Notace diagramu zahrnuje herce (actors), případ užití a asociace. Herec představuje roli, kterou zastupuje člověk (nebo jiný nelidský subjekt) s ohledem na systém. Roli může zastupovat mnoho lidí a naopak jeden člověk může zastupovat více rolí, proto je důležité myslet z hlediska aktérů a rolí spíše než z hlediska jedinců. (Vidgen, 2003) Souvislost mezi aktéry a případy užití lze vidět na obrázku 3.1, který naznačuje, že probíhá komunikace mezi hercem a případem užití, například při posílání / přijímaní zpráv.

Diagram užití používá tři typy asociací mezi případy užití, a to rozšíření, zahrnutí nebo generalizace. Rozšíření se používá tam, kde existují podobné případy užití, ale jeden nabízí více než ostatní. V našem případě (obr. 3.1) je to například prohlížení si jiných uživatelských profilů rozšiřující o možnost zaslat jim žádost o přátelství. Pokud více případů užití potřebují stejnou část funkcionality, pak se může tato funkcionality oddělit do samostatného případu užití a odkazovat na něj. V naší webové aplikaci (obr. 3.1) je to například přihlášení do administrace, které je potřeba pro správu aplikace. Generalizace je pak speciálním případem jiného případu užití.



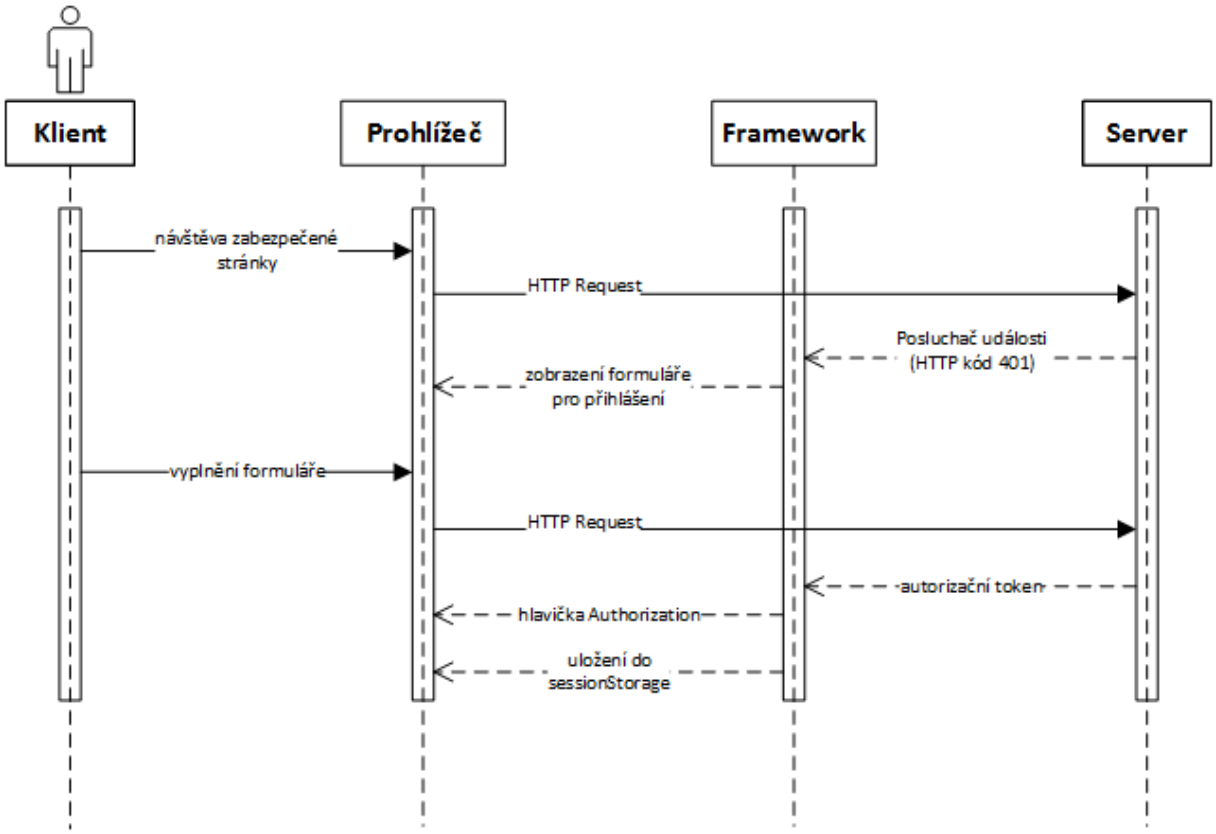
Obr. 3.1 Diagram užití zahrnující všechny aktéry webové aplikace

3.3.2 Sekvenční diagram

Sekvenční diagramy v UML se používají především k modelování interakcí mezi herci a objekty v systému a interakcí mezi objekty samotnými. UML obsahuje bohatou syntaxi pro sekvenční diagramy, která umožňuje modelovat mnoho různých druhů interakcí. Jak už název napovídá, sekvenční diagram představuje posloupnost interakcí, které se konají v průběhu určitého případu užití nebo jeho instanci. (Sommerville, 2011)

Objekty a herci zahrnutí do systému jsou uvedeni v horní části diagramu, s přerušovanou vertikální linií z nich vycházející. Interakce mezi objekty jsou označeny anotovanými šipkami. Obdélník ležící na vertikální přerušované linií značí „životnost“ daného objektu (například čas, ve kterém je instance objektu zapojena do určité procedury). Posloupnost

interakcí se reprezentuje shora dolů. Anotace šipek uvádějí volání objektů, jejich parametrů a návratových hodnot, zobrazené přerušovanou anotovanou šipkou.



Obr. 3.2 Sekvenční diagram přihlašování do aplikace

Příklad sekvenčního diagramu můžeme vidět na obrázku 3.2, který zobrazuje přihlašování do naší aplikace. Jako první nepřihlášený uživatel přijde na stránku, ze které se dotazujeme na zabezpečená data, prohlížeč pak zašle dotaz na API. Server zjistí, že se uživatel ptá na data, která jsou k dispozici pouze přihlášeným uživatelům, a odešle zpět HTTP kód 401. Aplikace zaregistruje, že byla vrácena tato chyba a vyšle zprávu všem registrovaným posluchačům uvnitř aplikace, že je vyžadováno přihlášení. Tuto zprávu zachytí framework, který zobrazí formulář pro přihlášení. Poté, co uživatel vyplní přihlašovací údaje a odešle formulář, server vrátí autentizační token, který framework uloží do sessionStorage prohlížeče a nastaví hlavičku Authorization pro příští požadavky.

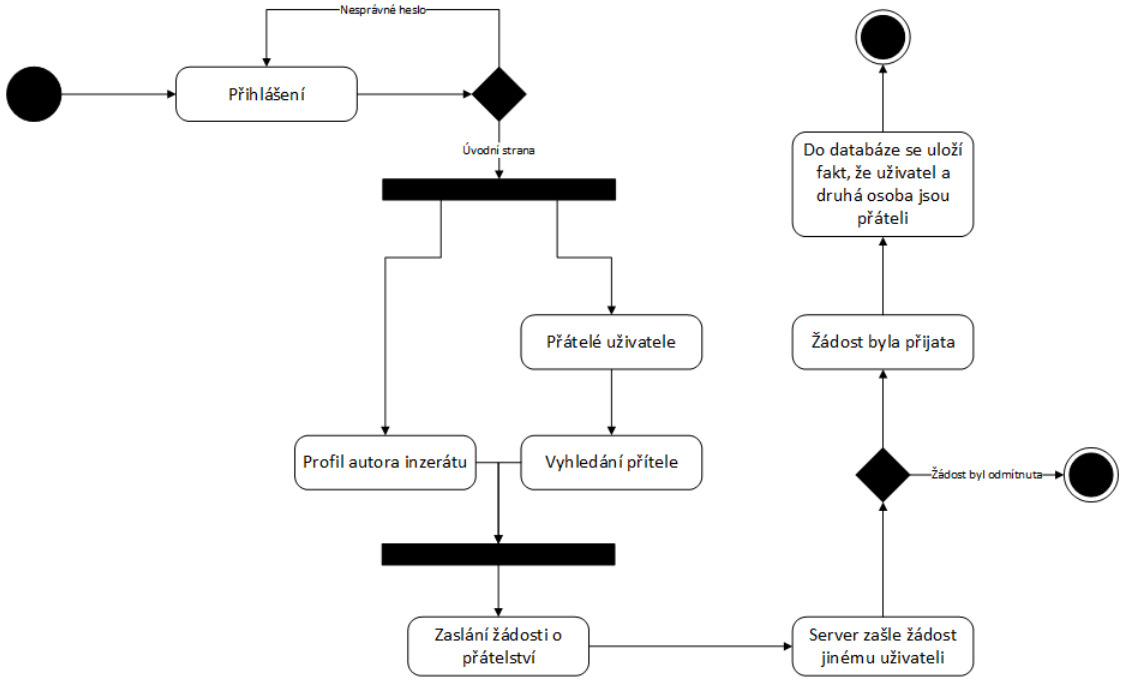
3.3.3 Diagram aktivit

Model-driven software development (MDSD), česky volně přeložené jako modelem řízený vývoj software, je nové vývojové paradigma softwaru. (France, 2006) Jeho výhodou je zvýšená produktivita s podporou vizualizace domén. V MDSD vývojáři používají návrhový model také pro testování software, zvláště pak pro objektově orientované programy. Tři hlavní důvody pro použití návrhového modelu v testování objektově orientovaného programu jsou: (1) tradiční metody testování softwaru berou v úvahu pouze statický pohled na kód, který však nemusí být dostačující pro testování dynamického chování objektově orientovaného systému (Binder, 2009), (2) testování kódu v objektově orientovaným systému je složitý a zdlouhavý proces. Naopak modelování může pomoci testerům software pochopit systém lepším způsobem a zjistit nedostatky pouze jednoduchým aplikováním modelu ve srovnání s kódem, (3) model založený na generování testů může být naplánovaný již v rané fázi vývoje životního cyklu softwaru a umožňuje tak paralelní kódování i testování. (Kundu, 2009)

Diagram aktivit patří mezi důležitý diagram podporovaný UML 2.0. Používá se pro modelování podnikových procesů, pro modelování řízení toků mezi objekty, při modelování složitých operací apod. Hlavní výhodou je jeho jednoduchost a snadnost porozumění toku logiky systému. Účelem diagramu aktivit je modelování procesního toku akcí, které jsou součástí větší aktivity. V projektech kde jsou použity případy užití, mohou diagramy aktivity modelovat konkrétní situaci na detailnější úrovni. Tyto diagramy mohou být použity nezávisle jak pro modelování procesů na obchodní úrovni, tak také pro modelování procesů na úrovni systému. Vzhledem k tomu, že diagram modeluje procesní tok, tak se více zaměřuje na sekvenci vykonávajících akcí a na podmínky, které spouštějí nebo hlídají tyto události. Diagram aktivit se také zaměřuje pouze na interní aktivity a ne na akce, které volají aktivity v jejich procesním toku nebo které spouštějí aktivity na základě určité události. (Bell, 2003)

Diagram aktivit v naší webové aplikaci můžeme vidět na obr. 3.3, konkrétně zaslání žádosti o přátelství. Po inicializaci procesu je třeba se přihlásit, v případě zadání špatných údajů je uživatel vyžádán o opětovné zadání údajů. Po přihlášení má uživatel dvě možnosti, které se dělí po forku. V prvním případě může jednoduše kliknout na autora určitého inzerátu a dostat se na jeho profilovou stránku, kde si zažádá o přátelství. Ve druhém případě si může rozkliknou svou stránku se seznamem aktuálních přátel a zde do vstupního pole zadat a vyhledat uživatele, kterému chce poslat žádost. Poté server zašle druhému uživateli tuto

žádost, kterou buď odmítne, nebo bude ignorovat a proces se ukončí, nebo žádost příjme a sytém na ní zareaguje uložením faktu, že jsou oba uživatelé přáteli.



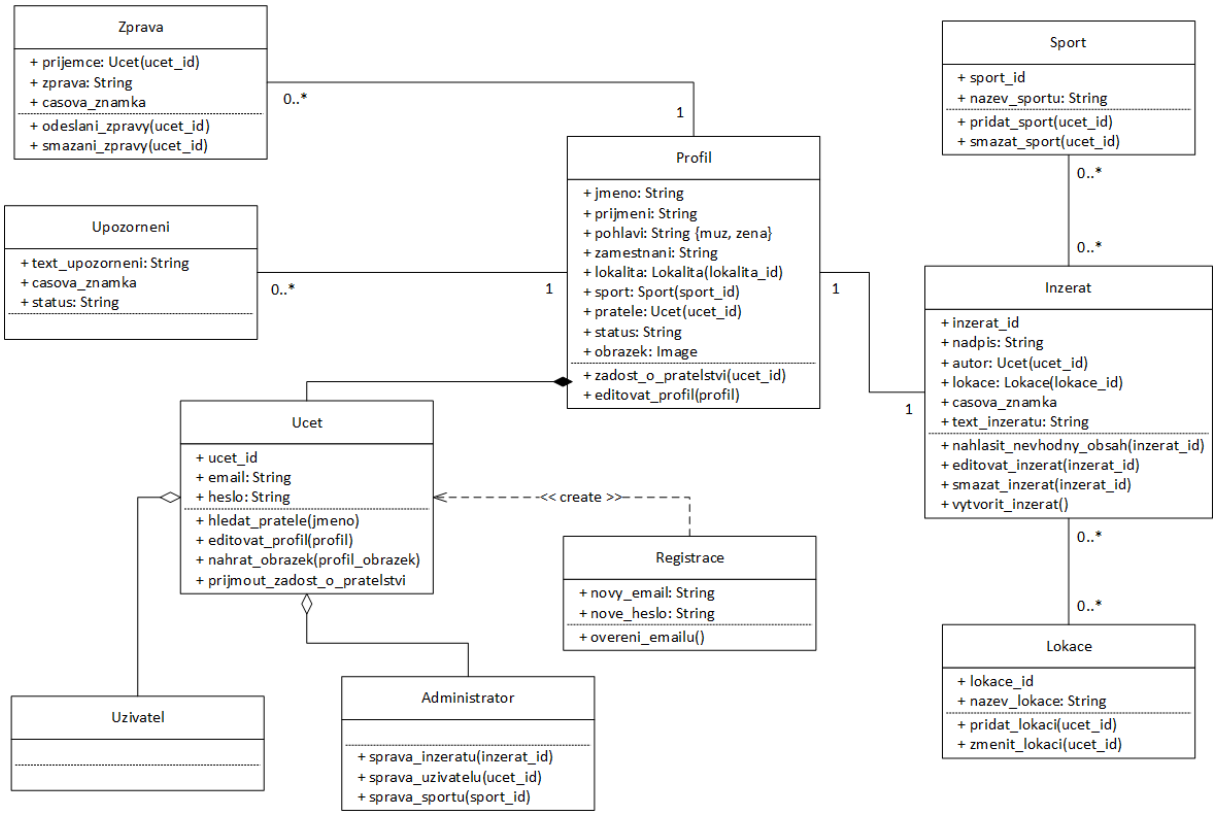
Obr. 3.3 Diagram aktivit při zaslání žádosti o přátelství

3.3.4 Diagram tříd

Účelem diagramu tříd je ukázat statickou strukturu systému, který je modelován. Diagram konkrétně zobrazuje entity v systému společně s jejich vnitřní strukturou a se vztahy s ostatními entitami v systému. Protože diagram tříd modeluje pouze statickou strukturu systému, tak jednotlivé instance nejsou zobrazeny. V naší aplikaci v diagramu zobrazeném na obr. 3.4 můžeme vidět například entitu sport, ale nezobrazuje již její instance jako například fotbal, horolezectví, vybíjená apod.

Diagram tříd typicky využívají vývojáři, protože mohou tak zjistit detaily o třídách, které jsou nebo budou naprogramovány, a o jejich atributech a metodách. Diagramy tříd jsou zvláště vhodné také pro modelování podnikových procesů. Podnikoví analytici mohou

používat diagramy k modelování podnikových krátkodobých aktiv a zdrojů, jako jsou například účetní knihy, produkty nebo zeměpisná hierarchie. (Bell, 2003)



Obr. 3.4 Diagram tříd zobrazující všechny entity v systému

Diagram tříd využívající naše webová aplikace je zobrazena na obrázku 3.4, kde se nachází celkem deset entit. Každá entita obsahuje název třídy, případně potřebné atributy a metody. V diagramu můžeme vidět také různé typy asociací mezi jednotlivými entitami. Entity Administrator a Uzivatel mají s entitou Ucet slabou formu agregace, což je zobrazeno binární spojením s prázdným symbolem diamantu u agregované entity. Přerušovaná čára mezi entitami Registrace a Ucet značí vztah závislosti. To znamená, že entita Registrace využívá entitu Ucet, ale neudržuje mezi nimi permanentní vztah. Závislost je navíc označena stereotypem <<create>> což označuje, že Registrace vytváří objekty (instance) typu Ucet. Nejčastější variantou asociace mezi entitami je kardinální, kde se značí multiplicita elementů. Multiplicita pak tedy obsahuje nezáporný počet instancí, které je možné vytvořit. Počet udává interval, který má nižní hranici a vyšší hranici (případně nekonečnou). Příklad kardinálního vztahu v diagramu tříd můžeme vidět na obr. 3.4 mezi entitami Profil a Zpráva. Jejich multiplicita udává, že jeden uživatel (respektive entita Profil) může napsat nula až nekonečno

zpráv, avšak entita Zpráva musí mít vždy jen jednoho uživatele (entitu Profil). U multiplicity je třeba však stále myslet na to, že omezení počtu se týká instancí entit.

3.4 Struktura a návrh REST API uživatelské části

Při tvorbě návrhu budeme hledat nejjednodušší řešení, které splní zadané požadavky. Je vhodné tedy uvést seznam stránek, které se budou zobrazovat v uživatelské části. Na těchto stránkách pak budeme zpracovávat několik operací, kvůli kterým bude potřeba komunikovat s API. Dále navrhneme samotné API z pohledu uživatelské části webu, kde každá z URL bude prefixovaná řetězcem `/api/`. Všechna uvedená URL se týkají pouze komunikace s API, nejde tedy o URL, která uvidí zákazník.

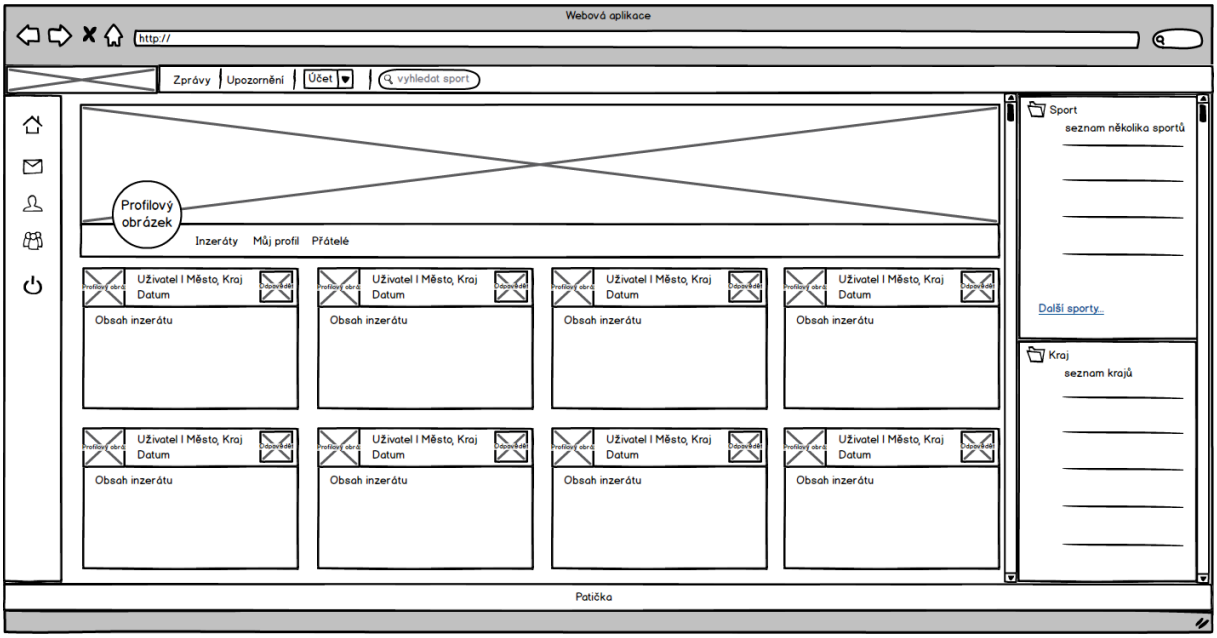
- úvodní stránka
    - Získání seznamu sportů pro menu
        - `GET /api/sport`
    - Získání seznamu regionů pro menu
        - `GET /api/region`
    - Získání všech inzerátu na hlavní stránku
        - `GET /api/adverts`
- seznam zpráv
    - získání všech zpráv přihlášeného uživatele
        - `GET /api/messages`
- detail zprávy
    - získání zpráv od určitého uživatele
        - `GET /api/messages/:username`
- profil uživatele
    - získání všech údajů daném uživateli
        - `GET /api/profil/:username`
    - úprava údajů přihlášeného uživatele
        - `PUT /api/profil/:username`
- úprava inzerátu
    - získání všech údajů o inzerátu daného uživatele, pokud existuje
        - `GET /api/advert/:username`
    - úprava údajů inzerátu přihlášeného uživatele
        - `PUT /api/advert/:username`

    - vložení nového inzerátu přihlášeného uživatele
        - `POST /api/advert/:username`
- detail sportu
    - získání všech inzerátů a jejich údajů, které spadají do daného sportu
        - `GET /api/sport/advert/:sportId`
- detail regionu
    - získání všech inzerátů a jejich údajů, které spadají do daného regionu
        - `GET /api/region/advert/:regionId`

3.5 Layout

Design webových stránek hraje velmi důležitou roli, protože je to první kontakt s uživatelem. Je třeba brát na vědomí, že komunikace s potenciálním uživatelem probíhá právě přes naše webové stránky, protože když uživatel navštíví webové stránky, s velkou pravděpodobností ví, proč tam je. Hlavní stránka by měla obsahovat všechny důležité elementy i bez posunování stránky a je třeba se vyvarovat ošklivému a neatraktivnímu designu. Webová stránky by měla mít své vlastní logo, název společnosti a obsahovat grafiku, která vypovídá o účelu stránek. První dojem při návštěvě stránky by měl být co nejlepší, uživatelovi totiž trvá rozhodnutí prvních deset až dvacet sekund, zda bude pokračovat v procházení stránek nebo ne. Zhruba pětina návštěvníků pak na stránkách zůstanou. (Nielsen, 2011)

Před samotným grafickým návrhem webu je důležité mít již vytvořenu tzv. logickou strukturu stránek, tzn. mít představu o účelu webu, o obsahu, rozdělení informací do menších logických celků, jejich vzájemnou provázanost apod. Grafický návrh hlavní stránky naší webové aplikace můžeme vidět na obr. 3.5. Obsahuje tří sloupce, které obsahují menu ve formě ikon, samotný obsah a seznam několika sportů s odkazem na další kategorie a seznam jednotlivých krajů České republiky. Se sloupci s inzeráty a se seznamy je pak možné posunovat. Návrh také obsahuje hlavičku, která obsahuje logo stránky, krátký výpis zpráv uživatele a jednotlivé upozornění, rozklikávací menu s účtem uživatele a poslední část je textové pole pro vyhledávání sportu. Patička bude obsahovat pouze název stránky a jednoduchý podtitulek.



Obr. 3.5 Grafický návrh webové stránky

4. Realizace internetové aplikace

Realizace internetové aplikace zahrnuje využití různých technologií a nástrojů, manipulaci s databází, operace na straně serveru a také na straně klienta a zobrazování dat přicházejících ze serveru. Před začátkem realizace nového projektu je vždy potřeba mít připravené všechny potřebné nástroje a mít představu o struktuře projektu, jelikož se jedná o velmi důležitou úlohu, protože změna struktury znamená velkou spotřebu času a s tím případně související rostoucí náklady. Použití vhodného frameworku nebo také „*stacku*“ za tímto účelem může zvýšit rychlost vývoje a usnadnit práci vývojáři.

Autoři MEAN stack na svých stránkách definují jako „*neústupný plnohodnotný JavaScriptový framework, který zjednodušuje a urychluje vývoj moderních webových aplikací.*“ (2015) MEAN je zaměřen na všechny typy JavaScript vývojáře, tedy na straně serveru i klienta. Jak už samotný název napovídá, jedná se o zkratku platforem, které jsou:

- **M**ongoDB
- **E**xpress
- **A**ngularJS
- **N**ode.js

Díky těmto čtyřem nejvíce používanými a vývojáři oceněnými technologiemi pro JavaScriptový vývoj, je položen základ pro tvorbu komplexních webových aplikací.

4.1 Horizontální a vertikální adresářová struktura

Ačkoliv je možné využít různých nástrojů, například Yeoman, MEAN.IO nebo MEAN.JS, které pomocí různých příkazů v CLI, tedy v příkazovém řádku, vygenerují potřebnou strukturu MEAN aplikace, je jednodušší vytvořit si vlastní adresářovou strukturu, která bude přesně odpovídat potřebám vývojáře, rychleji a snadněji se v ní bude orientovat a také je možné během vývoje využít nástroj Yeoman a přizpůsobit si ho podle své struktury a po té doplňovat strukturu o potřebné routy, controllery, pohledy či si rovnou vytvářet strukturu podle dané funkcionality.

Během vývoje nové aplikace často narazíme na problém, jak ji rozdělit na menší logické celky. Obecně JavaScript, a tedy i framework Express, je zcela nezávislý na jakoukoliv strukturu aplikace a je tedy možné dokonce vložit celou webovou aplikaci do jediného

JavaScriptového souboru. Je to z toho důvodu, že nikdo v minulosti neočekával, že by se ze JavaScriptu mohl stát plnohodnotný programovací jazyk. Protože je možné využít MEAN pro tvorbu všech druhů aplikací různých velikostí, je také možné navrhnout různou adresářovou strukturu. Struktura je obvykle úzce spjata s komplexností webové aplikace a rozlišují se dva hlavní typy, horizontální a vertikální struktura.

Vertikální struktura je založena na hierarchii složek a souborů podle funkcí, které daná aplikace implementuje, takže výsledkem je pak to, že každá funkcionalita má svou vlastní autonomní složku, která obsahuje MVC strukturu. Díky tomuto je vertikální struktura velmi vhodná pro větší aplikace, kde existuje téměř neomezené množství funkcí a každá funkce pak obsahuje nezbytný počet souborů. To umožňuje větším vývojovým týmům pracovat společně a udržovat každou funkcionalitu odděleně a taktéž je možné sdílet funkce mezi různými aplikacemi.

Druhým typem je horizontální struktura, která je založena na hierarchii složek a souborů podle jejich funkcionálních rolí spíše než podle funkcí, které implementují, což znamená, že všechny soubory jsou umístěny uvnitř hlavní aplikační složky, která obsahuje MVC strukturu. To také znamená, že existuje například jediná složka `controllers`, která obsahuje všechny controllery, jediná složky `models`, která obsahuje všechny modely atd. Horizontální struktura je vhodnější pro menší projekty, kde je počet funkcionalit omezený, takže soubory mohou být bez problémů umístěny do složek, které přestavují jejich roli.

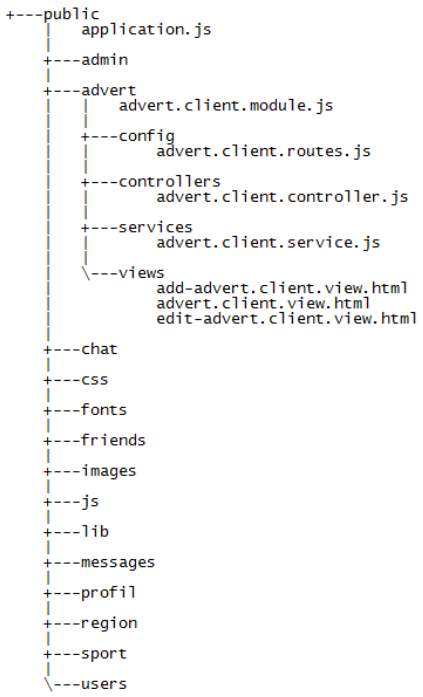
V naší webové aplikaci budou použity oba typy struktur pro demonstraci plné flexibility MEAN struktury. Na straně serveru bude použita horizontální adresářová struktura, která je zobrazena na obrázku 4.1.

```
+---app
|   +---controllers
|   +---models
|   +---routes
|   \---views
\---config
    |   config.js
    |   express.js
    |   mongoose.js
    |   passport.js
    |   socketio.js
    |
    +---env
    |       development.js
    |
    \---strategies
            facebook.js
            google.js
            local.js
            twitter.js
```

Obr. 4.1 Adresářová struktura na straně serveru

- Složka `app` obsahuje aplikační logiku frameworku Express a je rozdělena do následujících podadresářů, které reprezentují separaci funkcionalit v rámci MVC vzoru:
    - složka `controllers` obsahuje controllery Express aplikace,
    - složka `models` obsahuje modely Express,
    - složka `routes` obsahuje směrovací middleware Express aplikace,
    - složka `views` obsahuje pohledy Express aplikace.
- Složka `config` obsahuje konfigurační soubory Expressu aplikace. Zahrnuje jednotlivé moduly aplikace, které jsou umístěny v samostatných souborech.
    - složka `env` obsahuje jednotlivé vývojové prostředí konfiguračních souborů Express aplikace,
    - soubor `config.js` obsahuje samotnou konfiguraci Expressu,
    - soubor `express.js` obsahuje inicializaci Expressu,
    - soubor `mongoose.js` obsahuje inicializaci MongoDB a obsahuje všechny zaregistrované modely,
    - soubor `passport.js` obsahuje Passport middleware a serializaci a deserializaci uživatele při přihlašování. Taktéž zahrnuje strategie pro přihlašování k různým poskytovatelům či k lokálnímu účtu,
    - soubor `socketio.js` obsahuje inicializaci socket.io middleware, vytvoření sessions a jejich uložení do MongoDB a propojení s Passport middleware. Zahrnuje také veškerou komunikaci pomocí socket.io mezi klientem a serverem podle typu události.

Na straně klienta je použita vertikální adresářová struktura zobrazena na obrázku 4.2.



Obr. 4.2 Adresářová struktura na straně klienta

- Složka `public` obsahuje vše, co je potřeba k vykreslení webové aplikace klientovi. Je především rozdělena do jednotlivých separovaných složek podle dané funkcionality,
- soubor `application.js` obsahuje inicializaci AngularJS, definování základního modulu a seznam modulů potřebných pro dependency injection,
- složky `css`, `fonts`, `images` a `js` obsahují soubory, které klient uvidí při vykreslení webové stránky
- složka `lib` obsahuje veškeré AngularJS balíčky, které zde jsou nainstalovány a použity v aplikaci
- jednotlivé složky funkcionalit dále obsahuje podadresáře `config` se směrování, `controllers` s controllery, `services` s potřebnými službami, továrnami či poskytovateli a nakonec podadresář `views` s jednotlivými HTML soubory pro vykreslení dat. Ukázka rozdělení podadresářů je možné vidět na obrázku 4.2 pro funkcionalitu advert, tedy funkci zodpovědnou za práci s inzeráty.

Aplikace také v kořenovém adresáři obsahuje soubor `server.js` pro inicializaci a spuštění Express frameworku na portu 3000, soubor `package.json` jakožto seznam npm balíčků pro Node.js, soubor `bower.json` jakožto seznam balíčků pro AngularJS a nakonec soubor `.bowerrc` pro směrování instalace AngularJS balíčků do složky `lib`.

4.2 Konvence pojmenování

Protože MEAN aplikace mají často paralelní MVC strukturu komponent jak pro AngularJS tak také pro Express, tak je potřeba si stanovit určité konvence pro pojmenování jednotlivých souborů a nedošlo k chaotickému orientování podle funkcionalit co je na straně serveru a co na straně klienta.

Nejjednodušší variantou je přidat do názvu každé funkce také její funkcionální roli, takže například controller dané funkce bude mít název `funkce.controller.js`, model bude mít název `funkce.model.js` apod. Zde však nastává problém díky faktu, že MEAN aplikace využívá JavaScriptových MVC souborů pro AngularJS i pro Express a z toho důsledku bychom měli dva soubory téhož jména, například soubor `funkce.controller.js` může být použit pro AngularJS nebo také pro Express. Jako řešení bylo zvoleno přidat do názvu také místo, kde se daný soubor provádí. Díky tomuto přístupu pak můžeme soubory na straně serveru pojmenovávat jako `funkce.server.controller.js` a na straně klienta `funkce.client.controller.js`. Díky tomuto můžeme rychle identifikovat roli a místo vykonání potřebného souboru.

4.3 Inicializace a konfigurace Express aplikace

Pro inicializaci Express aplikace je potřeba jako první vytvořit soubor package.json, který definuje závislosti našeho projektu, respektive obsahuje názvy všechny frameworků či middleware, které jsou v projektu použity. K dodržení správné syntaxe však kromě názvu npm balíčku musí obsahovat také jeho verzi použitou v projektu a název a popis samotného projektu. Může však obsahovat mnoho dalších metadat jako jsou klíčová slova, bugy, licenci, enginy atd. Po vytvoření souboru package.json obsahující všechny námi potřebované npm balíčky, stačí v příkazovém řádku zadat příkaz `$ npm install` a vše se již automaticky nainstaluje do adresáře node_modules, obsahující všechny nainstalované balíčky připravené okamžitě k použití v aplikaci.

Framework Express nabízí poměrně jednoduchý konfigurační systém, který nám umožňuje do Express aplikace přidat určitou funkcionalitu. Ačkoliv existují předdefinované konfigurace, díky kterým můžeme měnit chování systému, tak také můžeme přidávat vlastní klíč / hodnota konfigurace, které budou odpovídat našim požadavkům. Další výhodou Express frameworku je možnost konfigurovat aplikaci podle toho, na jaké vývojovém prostředí

aplikace běží. Například pokud se chceme připojit k určitému MongoDB serveru, tak budeme potřebovat různé připojovací řetězce pro vývojové a produkční prostředí. Řešit by se to také dalo velkou řadou podmínek if/else, avšak to by bylo velmi náročné na údržbu. Jiným příkladem může být, když budeme chtít použít Express logger ve vývojovém prostředí a ne v produkčním, zatímco komprese těla odpovědi ze strany serveru je mnohem výhodnější v produkčním prostředí.

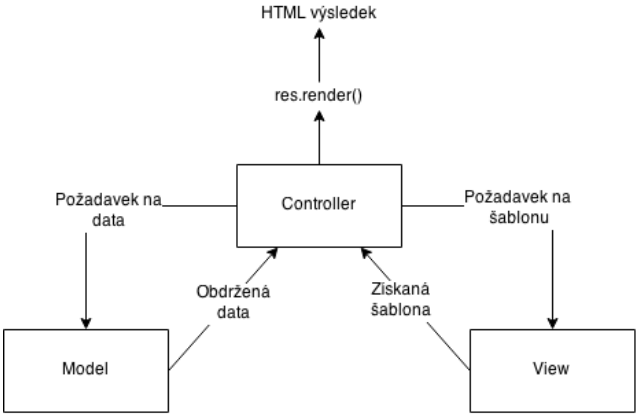
Abychom toho dosáhli, tak potřebujeme použít vlastnosti `process.env`, což je globální proměnná, která nám umožňuje získat přístup k předdefinovaným proměnným prostředí, nejčastěji k proměnné `NODE_ENV`. Tato proměnná prostředí se pak často používá pro definování konfigurací pro různá prostředí. Díky tomu pak v naší webové aplikaci můžete napsat toto:

```
if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
} else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
}
```

Jak můžeme z kódu vidět, tak proměnná `process.env.NODE_ENV` determinuje naše prostředí v jakém běží aplikace a podle toho konfiguruje Express aplikaci. Jednoduše jsme použili metodu `app.use()` pro načtení middleware morgan ve vývojovém prostředí, který při každém načtení stránky loguje všechny soubory, včetně těch změněných, a taktéž vrací kód odpovědi. Naopak v produkčním prostředí Express načte middleware compress, který vrací zkompresované tělo odpovědi.

4.3.1 Vykreslení Express pohledů

Běžnou funkcí webových frameworků je schopnost vykreslovat pohledy. Základní myšlenkou je vložení dat do enginu šablony, který vykreslí výsledný pohled, nejčastěji v HTML. V MVC vzoru controller používá model k získání dat a šablonu pohledu pro vykreslení HTML výsledku jak je zobrazeno na obrázku 4.3.



Obr. 4.3 MVC vykreslení výsledku (Zdroj: vlastní)

Express má dvě metody pro vykreslení pohledů. První je `app.render()`, která se používá k vykreslení pohledu a poté použití HTML v callbacku. Druhá metoda `res.render()` je běžnější a vykresluje pohled lokálně a poté posílá HTML jako odpověď.

Framework Express nabízí řadu Node.js šablonových enginů, které plní tuto funkcionalitu. Jako výchozí šablonový engine Expressu je nastaven Jade, který má ale poměrně odlišnou strukturu od klasického HTML a proto je v naší aplikaci použit šablonový engine EJS, který zachovává strukturu HTML a pomocí speciálních značek je možné v šabloně vykreslovat i proměnné, které si definujeme v patřičném controlleru.

Při vykreslování pohledů potřebujeme také zobrazovat statické soubory, jako jsou například obrázky, JavaScriptové soubory třetích stran, CSS soubory či fonty. Express pro tento účel nabízí middleware `express.static()`, který nastavuje cestu, odkud se budou statické soubory načítat:

```
app.use(express.static('./public'));
```

4.4 Manipulace s Mongoose

Mongoose je populární Node.js ODM modul, který zajišťuje podporu MongoDB v naší aplikaci. Mongoose používá schéma pro modelování entit, nabízí předdefinované či vlastní validaci, je možné definovat virtuální atributy a využívat middleware pro různé operace. Cílem mongoose je tak vyplnit mezeru mezi MongoDB, který nevyužívá žádné schéma, a potřebou vytvářet entity reálného světa.

4.4.1 Mongoose schéma

Mongoose je tedy Node.js modul, který poskytuje schopnost modelovat objekty a uložit je jako MongoDB dokumenty. Po nainstalování MongoDB a mongoose jakožto npm balíčku definovaném v souboru package.json, je třeba se k němu připojit pomocí URI, což je URL řetězec, který říká ovladači MongoDB jak se připojit k instanci databáze. MongoDB URI má pak obvykle tento tvar:

```
mongodb://uzivatel:heslo@host:port/databaze
```

Ačkoliv je možné URI uložit přímo do konfiguračního souboru frameworku Express, je lepší využít konfigurační soubor, který můžeme pro každé prostředí definovat jiný.

MongoDB používá kolekce k ukládání dokumentů, které nevyžadují, aby dokumenty měli stejnou strukturu. Nicméně když pracujeme s objekty, je často praktické, aby si dokumenty byly podobné. Mongoose využívá objektu `schema` k definování seznamu atributů, které dokument musí nebo by měl obsahovat, a každý atribut má pak svůj typ, případně další omezení. Poté, co je specifikované schéma, je potřeba definovat konstruktor `model`, který se používá k instanci MongoDB dokumentů. Například pro funkcionalitu zpráv v naší aplikaci definujeme toto schéma:

```
var MessageSchema = new Schema({
    created: {
        type: Date,
        default: Date.now
    },
    message: {
        type: String,
        default: '',
        required: 'Zpráva nesmí být prázdná'
    },
    from: {
        type: String
    },
    to: {
        type: String
    }
});
mongoose.model('Message', MessageSchema);
```

Z kódu je patrné definování celkem čtyřech atributů, které mají vlastní typy a případně další elementy. Atribut created definuje datový typ a jako výchozí hodnota je aktuální den a čas, tedy při vkládání nového dokumentu není potřeba tento atribut nijak specifikovat, protože se vloží automaticky. Atribut message může obsahovat řetězec, avšak musí být explicitně definován, jinak se dokument do databáze neuloží a vyhodí hlášení o chybějícím atributu. Nakonec můžeme vidět konstruktor `model`, který vytváří novou instanci s názvem Message.

4.4.2 CRUD operace

Mongoose pro práci s dokumenty nabízí díky svému API řadu metod, které obecně poskytují CRUD operace, tedy vytváření nových záznamů (**c**reate), čtení záznamů (**r**ead), aktualizování záznamů (**u**pdate) a mazání záznamů (**d**elete). Práce s modely obstarávají jednotlivé controllery podle jejich funkcionality, aby se tak dodržela adresářová struktura webové aplikace.

Pokud bychom chtěli například vytvořit nový dokument ve schématu Sport, stačí v controlleru pro jeho obsluhu vytvořit tento kód:

```
exports.create = function (req, res) {
    var sport = new Sport(req.body);
    sport.save(function (err) {
        if (err) {
            return res.status(400).send({
                message: getErrorMessage(err)
            });
        } else {
            res.json(sport);
        }
    });
};
```

V tomto kódu jsme prvně definovali metodu `create()` a pomocí klíčového slova `new` vytváříme novou instanci modelu Sport, která je rozšířená o data přicházející z požadavku. Nakonec díky instanci zavoláme metodu `save()`, která buď uloží nový objekt do databáze a vypíše výsledek v JSON formátu, nebo se uložení nepodaří a vrátí se chyba.

Metoda `find()` je metoda modelu díky které můžeme získat více dokumentů uložených ve stejné kolekci a jedná se o Mongoose implementaci MongoDB `find()` metody. Pokud bychom chtěli tedy například získat všechny sporty, můžete využít této metody:

```
exports.list = function (req, res) {
    Sport.find({}).sort('nazev').exec(function (err, sports) {
        if (err) {
            return res.status(400).send({
                message: getErrorMessage(err)
            });
        } else {
            res.json(sports);
        }
    });
};
```

Je třeba si všimnout jak nová metoda `list()` využívá metody `find()` k získání pole všech dokumentů v kolekci sports. V tomto případě metoda `find()` využívá dva argumenty, MongoDB objekt a callback, avšak může akceptovat až čtyři parametry – samostatný MongoDB dotaz, pole dokumentu které chceme vrátit, další možnosti dotazu jako například `skip` nebo `limit`, a nakonec callback. Pokud bychom chtěli vrátit pouze jeden dokument, můžeme využít metody `findOne()`.

Mongoose model nabízí několik dostupných metod pro potřebu aktualizace již existujícího dokumentu. Mezi ně patří metody `update()`, `findOneAndUpdate()` a `findByIdAndUpdate()`. Každá z těchto metod obsluhuje jinou úroveň abstrakce, zajišťující co nejjednodušší aktualizaci. V případě, že bychom chtěli dokument smazat, je možné díky mongoose modelu využít metod `remove()`, `findOneAndRemove()` a `findByIdAndRemove()`.

4.5 Express RESTful API

Protože Express aplikace bude sloužit hlavně jako RESTful API pro AngularJS aplikaci, je vhodné si vytvořit směrování podle REST principů. Každý objekt je dostupný pomocí unikátního identifikátoru zdroje, se kterým můžeme pracovat díky čtyřem základním metodám: GET, POST, PUT a DELETE. A právě pro každé URI ve směrování definujeme ty operace, které naše webová aplikace bude potřebovat. Například směrování pro operace s inzeráty může vypadat takto:

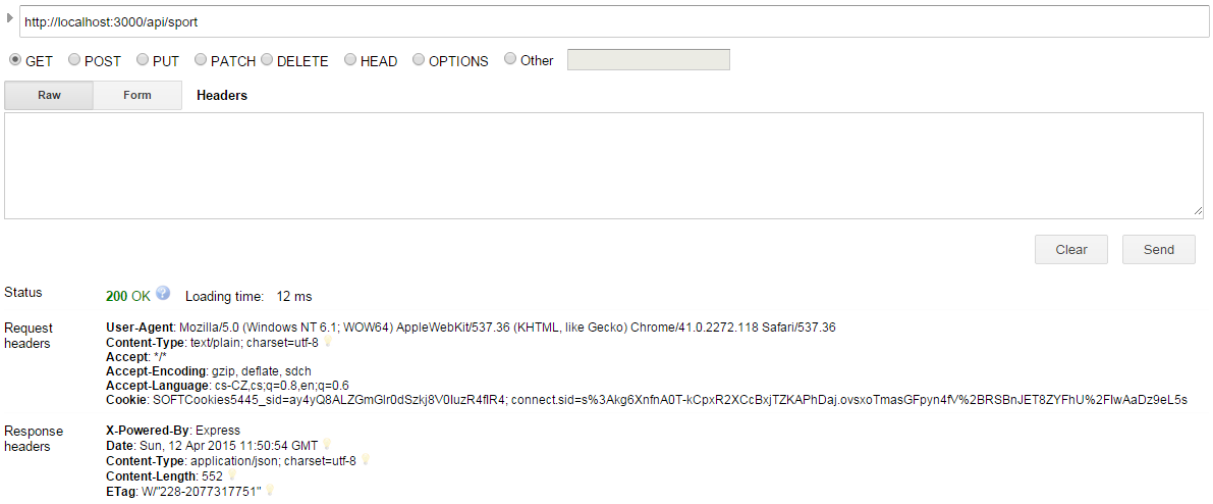
```
module.exports = function (app) {
    app.route('/api/advert')
        .get(advert.list)
        .post(advert.create);

    app.route('/api/advert/:username')
        .get(advert.read)
        .post(advert.create)
        .put(advert.update)
        .delete(advert.delete);
    app.param('username', advert.advertByID);
};
```

První unikátní adresa jak je možné pracovat s inzeráty je `/api/advert`, kde `api/` značí právě naše API rozhraní, se kterým je možné operovat i z vnějšku aplikace. Na této adrese můžeme pomocí HTTP metody `get()` získat všechny záznamy, nebo naopak pomocí metody `post()` vložit zcela nový záznam. Parametr v těchto metodách pak obsahuje middleware, které se mají zavolat v případě dané HTTP metody.

Druhá adresa již pracuje s inzerátem daného uživatele podle jeho uživatelského jména. V Express frameworku přidání dvojtečky před slovo značí pro směrování, že toto slovo bude bráno jako parametr požadavku. Využíváme zde i metodu `app.param()`, která definuje middleware, který se vykoná dříve než kterýkoliv jiný middleware, který využívá tento parametr. V našem případě se tak jako první zavolá middleware `advertByID()`, která najde požadovaný inzerát, výsledek vloží do požadavku `req.advert` a až poté se zavolá middleware `read()`, který daný požadavek `req.advert` odešle zpět v JSON formátu. Zbylé middleware `post()`, `put()` a `delete()` pak fungují klasicky podle HTTP požadavku.

Pro otestování našeho RESTful API můžeme použít několik známých nástrojů. Jedním z nich je cURL z příkazového řádku, avšak je poměrně náročný na psaní a jeho úpravu. Jiným řešením je využít některého z řady aplikací rozšíření pro Google Chrome. Těmi nejrozšířenějšími jsou Postman a Advanced REST client. Právě druhý zmíněný nabízí o něco lepší a jednodušší uživatelské rozhraní a proto byl vybrán pro testování aplikace z pohledu REST API. Jak je možné vidět na obrázku 4.4, v aplikaci Advanced REST client je po zadání adresy možné vybrat potřebnou HTTP metodu, v případě potřeby pomocí formuláře vložit hodnoty do požadavku a po odeslání získáme odpověď ze strany serveru včetně příjemně zformátovaného výsledku v JSON formátu.



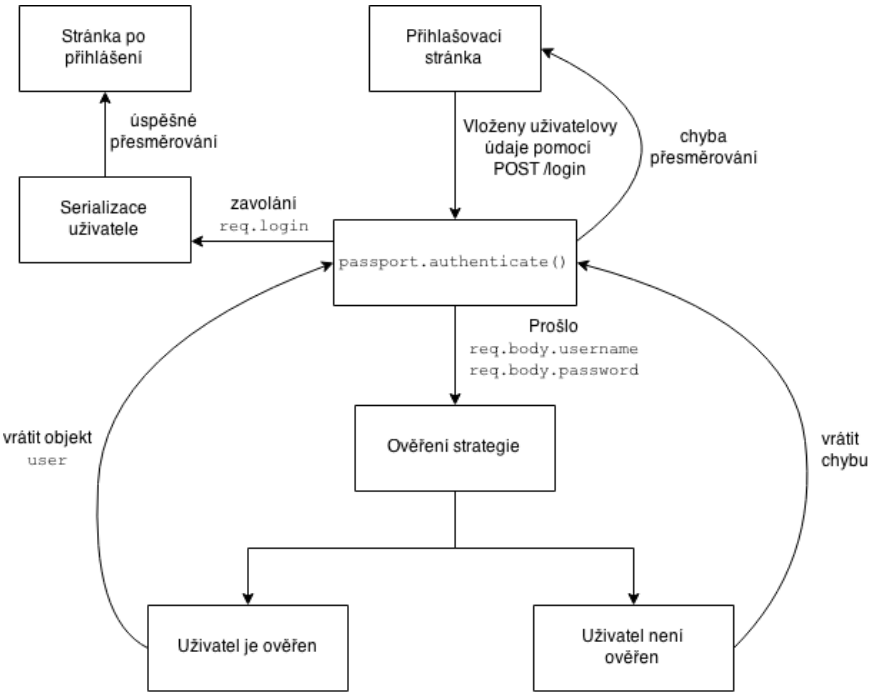
Obr. 4.4 Uživatelské rozhraní aplikace Advanced REST client

4.6 Přihlašování pomocí Passport

Autentizace je jednou z nejdůležitější části naší webové aplikace. Zajištění registrace a přihlášení uživatele je důležitou funkcí, která však může občas představovat nepříjemné zdržení. Express jakožto samostatný npm balíček, tuto funkci postrádá a tak je třeba využít jiného npm balíčku. Passport je Node.js modul, který se používá jako middleware při požadavcích na přihlášení. Modul nabízí různé autentizační metody využívajíc mechanismu zvaného strategie, který umožňuje implementovat komplexní autentizační vrstvu se zachováním přehledného kódu. Před jeho použitím je již tradičně nutné ho pomocí npm nainstalovat a poté k němu doinstalovat další moduly představující různé autentizační strategie. Po instalaci pak je potřeba Passport inicializovat a taktéž zaregistrovat middleware `passport.session()`, který využívá Express session pro sledování uživatelského session.

Aby bylo možné nabídnout různé autentizace, Passport používá oddělené moduly, které implementují různé autentizační strategie. Každý modul tedy poskytuje různé autentizační metody, jako jsou uživatelské jméno / heslo nebo OAuth autentizaci. V naší aplikaci to pak funguje tak, jak je zobrazené na obrázku 4.5. Tedy poté, co uživatel odešle své přihlašovací údaje, odešle se POST požadavek na middleware `passport.authenticate()`, který zavolá implementaci dané strategie. Passport převezme údaje z `req.body.username` a `req.body.password` a vloží je do verifikační funkce dané strategie. V případě, že se jedná o lokální strategii, tak se načtou databázové údaje o uživateli a porovnají se hesla. V případě, že výskytu chyby při naší interakci s databází, je nutné vrátit callback `done(err)`. Pokud

nemůžeme najít uživatele nebo hesla nesouhlasí, vrátíme callback `done(null, false)`. A pokud všechno proběhlo v pořádku a chceme uživatele přihlásit, vrátíme callback `done(null, user)`. Zavolání callbacku done se vrátíme zpět `passport.authenticate()` a pokud je uživatel přihlášen, tak middleware zavolá funkci `req.login()`. To vyvolá metodu `passport.serializeUser()`, která má přístup k objektu uživatele, které jsme předtím předali middleware. Poté se rozhodne, která data se uloží do session jako `req.session.passport.user`. Tento výsledek je pak také připojen k požadavku jako `req.user` a uživatel je přesměrován na stránku po úspěšném přihlášení, v našem případě na domovskou stránku.



Obr. 4.5 Přihlašování pomocí Passport (Zdroj: vlastní)

Protože při registraci není vhodné ukládat do databáze heslo v čisté textové podobě, upravíme si mongoose model User:

```
UserSchema.pre('save', function (next) {
    if (this.password) {
        this.salt = new
            Buffer(crypto.randomBytes(16).toString('base64'), 'base64');
        this.password = this.hashPassword(this.password);
    }
    next();
});
```

```
UserSchema.methods.hashPassword = function (password) {
    return crypto.pbkdf2Sync(password, this.salt, 10000,
        64).toString('base64');
};

UserSchema.methods.authenticate = function (password) {
    return this.password === this.hashPassword(password);
};
```

Prvně si vytvoříme pomocí `pre-save` middleware, tedy funkcí, která se vykoná ještě před samotným uložením uživatele, pseudo-náhodnou hashovací sůl a poté nahradíme uživatelovo heslo zahashovaným heslem za použití instance metody `hashPassword()`. Druhá metoda `authenticate()` pak akceptuje heslo jako parametr, zahashuje ho a porovná se současným uživatelovým heslem.

OAuth je autentizační protokol, který umožňuje uživatelům se registrovat do naší webové aplikace pomocí externích poskytovatelů bez potřeby vkládat na naší přihlašovací stránce jejich přihlašovací údaje. OAuth je převážně využíván sociálními platformami, jako jsou Facebook, Google nebo Twitter, kteří umožňují se přihlásit do naší aplikace pomocí jejich sociálnímu účtu. Passport poskytuje jednotlivé moduly pro tyto sociální služby a tím se tak vyhneme obtížné a komplexní implementaci tohoto mechanismu. Oficiální stránky Passportu nabízejí detailní popis jak tyto moduly použít, nicméně velmi důležité je také zaregistrovat se u těchto sociálních poskytovatelů. Díky tomu, že u nich zaregistrujeme naší aplikaci, získám také pro každou službu zvlášť jedinečné ID a klíč vygenerovaný jen pro naší webovou aplikaci.

Výsledkem je pak to, že můžeme využívat celkem čtyři různé Passport strategie – lokální a přihlašování pomocí účtu na Facebook, Google nebo Twitter.

4.7 Konfigurace a struktura AngularJS

AngularJS je frontendový JavaScriptový framework navržen pro tvorbu single-page aplikací s využitím MVC architektury. Přístupem AngularJS je rozšířit funkcionalitu HTML použitím speciálních atributů, které vážou JavaScriptovou logiku s HTML elementy. Tato schopnost umožňuje čistší DOM manipulaci skrze šablonu na straně klienta a poskytuje také obousměrné svázání dat, které synchronizuje model a pohled. AngularJS také zlepšuje kódovou strukturu a využívá dependency injection.

Při instalaci AngularJS využíváme balíčkového manažera Bower, který slouží ke stejnému účelu jako soubor package.json, avšak na straně klienta. Díky příkazu `$ bower install` pak můžeme nainstalovat všechny závislosti, které bower obsahuje.

Každá AngularJS aplikace vyžaduje při inicializaci alespoň jeden modul pomocí metody `angular.module()`. Když tuto metodu zavoláme s jediným argumentem, získáme existující modul s tímto jménem, pokud takový modul neexistuje, vyhodí se chyba. Nicméně tuto metodu zavoláme s více argumenty, AngularJS vytvoří modul s tímto jménem, jeho závislosti a konfigurační funkci, která se zavolá v okamžiku zaregistrování tohoto modulu.

4.8 Implementace AngularJS MVC entit

První entita MVC architektury je view, neboli pohled. AngularJS pohledy jsou HTML šablony vykreslené AngularJS kompilátorem pro manipulaci s DOM na stránce.

Druhým prvkem jsou controllers, které AngularJS používá k vytvoření nové instance objektu controlleru. Jejich účelem je rozšířit datový model pomocí objektů zvaných scope. Díky tomu můžeme scope definovat jako spojovatel mezi pohledem a controllerem. Díky tomuto objektu může controller manipulovat s modelem, který automaticky zobrazuje tyto změny do pohledu a naopak.

Abychom plně využili MVC architektury, je potřeba zajistit kontrolu nad URL směrováním. Pro tento účel existuje modul ngRoute, který umožňuje definovat URL adresu a její odpovídající šablonu, která se vykreslí kdykoliv když uživatel navštíví tuto adresu. Díky tomu, že AngularJS je single-page framework, ngRoute zajišťuje kompletní směrování v prohlížeči. To znamená, že namísto získání webových stránek ze serveru AngularJS načte definovanou šablonu, zkompiluje ji, a umístí výsledek do konkrétního DOM elementu. Server tak obsluhuje šablony jen jako statické soubory, avšak nereaguje na to, když se mění URL adresa. Díky tomu je tak Express server více orientován na API.

Výchozím chováním modulu ngRoute je vkládat do URL adresy znak hashe # jako součást směrování. Díky tomu, že je zvykem používat hash znak jako odkaz v rámci samotné aplikace, tak prohlížeč nevytváří požadavek na server. Výsledkem je tedy to, že běžné směrování v AngularJS má tvar `http://localhost:3000/#/inzeraty`. Nicméně single-page aplikace mají jeden hlavní problém a to, že nejsou indexovatelné vyhledávacími roboty a trpí pak nízkým SEO. Řešením je možnost, kterou nabízejí vyhledávací enginy, tedy označit naší aplikaci jako SPA. Díky tomu budou vyhledávací roboti vědět, že aplikace využívá

AJAX pro vykreslení nové cesty a mají chvíli počkat pro získání výsledku. Abychom aplikaci označili jako SPA, tak potřebujeme použít tzv. hashbangů. Ty jsou implementovány jako přidání vykřičníku hned za znakem hashe, takže stejná adresa pak bude mít tvar `http://localhost:3000/#!/inzeraty`.

4.8.1 Služby AngularJS

Služby AngularJS jsou entity obvykle používané ke sdílení informací mezi různými entitami stejné AngularJS aplikace. Služby mohou využity k získání dat ze serveru, sdílet cachovaná data nebo injektovat globální objekty do jiných komponent. Existují dva typy služeb, předdefinované služby a vlastní služby.

Předdefinované služby slouží k abstrahování rutinních vývojových úloh. Obecně užívané služby jsou:

- `$http` – služba, která se stará o AJAX požadavky,
- `$resource` – služba, která se stará o RESTful API,
- `$location` – služba, která zajišťuje URL manipulaci,
- `$q` – služba, která se stará o tzv. promises,
- `$rootScope` – služba, která vrací kořenový objekt scope.

V případě, že bychom vytvořili obálku nad globálními objekty například za účelem sdílení kódu, je vytvoření vlastní služby nepostradatelnou částí AngularJS aplikace. Služby můžeme vytvořit pomocí tří metod - `provider()`, `service()` a `factory()`. Každá z těchto metod umožňuje definovat název služby a funkci, která slouží pro různé účely.

4.8.2 Modul ngResource

Služba `$http` může sloužit jako komunikační vrstva mezi AngularJS aplikací a backend API, nicméně `$http` služba poskytuje vývojářům nízko úrovňové rozhraní při práci s HTTP požadavky a pro práci s RESTful API tak není moc vhodná. Za tímto účelem byl vytvořen modul ngResource, který poskytuje vývojářům jednoduchou cestu při komunikaci s RESTful zdrojem.

Modul ngResource nabízí nový typ služby `factory()`, která může být injektovaná do AngularJS entity. Služba `$resource` používá základní URL cesty a také nabízí možnost

konfigurovat si některé vlastnosti podle potřeby. Abychom mohli využívat modulu ngResource, musíme zavolat metodu služby `$resource`, která vrací `$resource` objekt. Služba akceptuje čtyři argumenty:

- `Url` – URL adresa s případným parametrem s prefixem dvojtečky, například `api/advert/:username`,
- `ParamDefaults` – výchozí hodnoty pro URL parametry, které mohou zahrnovat hodnoty nebo řetězec s prefixem @, takže hodnota parametru je extrahovaná z objektu,
- `Actions` – metody, které můžeme použít pro rozšíření výchozích akcí zdrojů,
- `Options` – volitelná možnost rozšířit výchozí chování `$resourceProvider`.

Návratový objekt ngResource obsahuje několik metod pro manipulaci s RESTful zdroji a mohou být případně rozšířeny vlastními metodami. Výchozími metodami zdrojů jsou:

- `get()` – využívá HTTP metody GET a očekává JSON objekt jako odpověď,
- `save()` - využívá HTTP metody POST a očekává JSON objekt jako odpověď,
- `query()` - využívá HTTP metody GET a očekává JSON pole jako odpověď,
- `remove()` - využívá HTTP metody DELETE a očekává JSON objekt jako odpověď,
- `delete()` - využívá HTTP metody DELETE a očekává JSON objekt jako odpověď.

Zavolání kterékoliv z těchto metod zavolá použití $http služby a vyvolá HTTP požadavek se specifickou HTTP metodou, URL a parametry. Metoda instance `$resource` pak vrátí prázdnou referenci na objekty, který se naplní, jakmile se vrátí data ze serveru. Jako příklad můžeme uvést vlastní službu Friends, která má za úkol komunikovat s API na straně serveru:

```
angular.module('friends').factory('Friends', ['$resource',
    function ($resource) {
        return $resource('api/friends/:username', {
                username: '@username'
            },
            {
                update: {
                    method: 'PUT'
                }
            },
```

```
            {
                'query': {
                    method: 'GET',
                    isArray: true
                }
            }
        );
    }]);
```

Služba `factory()` s názvem Friends vrací `$resource` s požadovanou URL adresou na RESTful API a přidává k tomu i parametr username, který je z objektu automaticky extrahován. Také definovaná vlastní metoda update, která při zavolání automaticky využívá HTTP metody PUT. Poslední částí je použití `query()` namísto klasického `get()`, protože jako odpověď očekáváme pole namísto objektu. V controlleru bychom pak tuto službu využili velmi jednoduše. Po její registraci jakožto dependency injection můžeme naplnit $scope, který následně použijeme v příslušném pohledu, například takto:

```
$scope.findFriends = function () {
    Friends.query({
        username: $routeParams.username
    }, function (callback) {
        $scope.friends = callback[0];
    });
};
```

kde jako $routeParams využíváme parametr nacházející se v URL, například `api/friends/:johnDoe`.

4.9 Real-time pomocí Socket.io

Při realizaci naší webové aplikace běžící v reálném čase je použit modul Socket.io, který je použit k vytvoření nové instance Socket.io serveru, který bude komunikovat se socket klienty. Objekt serveru podporuje jak samostatnou implementaci, tak také schopnost využít ho společně s Express frameworkem. Instance serveru pak rozšiřuje sadu metod, které umožňují další operace s Socket.io serverem.

Když chce klient komunikovat se Socket.io serverem, tak je prvně odeslaný handshake HTTP požadavek. Server poté analyzuje požadavek k získání nezbytných informací pro

nadcházející komunikaci. Poté se podívá na konfigurační middleware, který je registrován se serverem, a vykoná ho ještě dříve, než proběhne samotné připojení. Jakmile je uživatel úspěšně připojen k serveru, je vykonán posluchač události při připojení a je vytvořen nová instance socketu. Když je pak handshake proces u konce, je klient přihlášen k serveru a veškerá komunikace probíhá pomocí socketové instance.

K zajištění komunikace mezi klientem a serverem využívá Socket.io strukturu, která kopíruje WebSockets protokol. Existují dva typy událostí – systémové události, které zobrazují status socket připojení, a vlastní události, které použijeme pro implementaci naší logiky.

Systémové události na straně serveru jsou například:

- `io.on('connection', ...)` – emitován, když je připojen nový socket,
- `socket.on('message', ...)` – emitován, když je odeslána zpráva pomocí metody `socket.send()`,
- `socket.on('disconnect', ...)` – emitován, když je socket odpojen.

Socket.io budeme v naší aplikaci používat především při práci se zprávami, konkrétně když uživatel pošle zprávu jinému uživateli. Uživatel má mimo jiné možnost svému příteli poslat zprávu hned, jakmile ho uvidí, že se přihlásil do naší aplikace. K tomu, aby viděl, že je online, můžeme použít právě systémovou událost:

```
var clients = {};
io.on('connection', function (socket) {
    console.info('New client connected (id=' + socket.id + ').');
    clients[socket.request.user.username] =
    {
        username: socket.request.user.username,
        socket_id: socket.id,
        status: 'online'
    };
}
```

Z kódu lze vidět, že jakmile se uživatel připojí k serveru, vypíše se uživatelovo ID socketu a poté se do listu podle jeho uživatelského jména uloží je ID socketu a status, že je online. Naopak při odhlášení se jeho status změní na offline.

Zatímco systémové události nám pomáhají při řízení připojení, v naší aplikaci budeme více využívat vlastní události. Socket.io nabízí dvě metody, obě jak na straně serveru, tak na

straně klienta. První metoda je `on()`, která zachytává události a druhá metoda je `emit()`, která naopak „odpaluje“ události mezi objekty serveru a klienta.

Implementace on() metody na straně serveru je poměrně jednoduchá záležitost, například:

```
socket.on('obdrzenaZprava', function (message) {
    io.emit('aktualizujZpravy', message);
});
```

kdy jsme svázali posluchače události s události obdrzenaZprava. Tato událost je pak zachycena v okamžiku, kdy klient emituje událost obdrzenaZprava. Naopak při vykonání této události je emitovaná událost aktualizujZpravy společně s objektem message, kterou musí zachytit posluchač na straně klienta, taktéž podobným způsobem:

```
Socket.on('aktualizujZpravy', function (message) {
    $scope.messages.push({
        "message": message.message,
        "from": message.from,
        "to": message.to,
        "created": message.created
    });
});
```

Při zachycení této události se pak do proměnné messages vloží nová zpráva obsahující samotnou zprávu, od koho a komu je určena a datum a čas poslání. Pohled na to zareaguje a klientovi se bez jakékoliv nutnosti obnovovat stránku zobrazí daná zpráva, která mu byla určena.

5. Závěr

Hlavním cílem této práce bylo analyzovat postupy a techniky sloužící k tvorbě webových aplikací pracujících v reálném čase a poté implementovat systém, který bude těchto poznatků využívat. Tento cíl byl úspěšně naplněn, kde účelem této práce nebylo vytvořit „tu nejlepší“ sociální aplikaci, protože díky velké řady konkurence toho není možné dosáhnout, ale spíše se snaží popsat návrh a realizaci moderní webové aplikaci pomocí nových technologií s důrazem na její jednoduchost, znovupoužitelnost a možnost dalšího rozšiřování. Při jejím vývoji byla snaha dodržet všech REST principů, nabídnout externím službám jednoduché API a pomocí Socket.io umožnit aplikaci běžet v reálném čase, tedy bez nutnosti obnovovat stránku při určité události.

V první kapitole je v krátkosti popsán vznik v současnosti jednoho z nejpopulárnějších programovacích jazyků a je uveden MEAN stack pro vývoj moderních internetových aplikací, takzvaných Single-page aplikací. Byl uveden cíl diplomové práce a popis sociální služby, kterou bude nabízet.

Druhá kapitola této diplomové práce popisuje teoretické východiska aplikace, které popisují technologie využité ve vývoji webové aplikace. Také jsou zde popsány používané nástroje a další knihovny, které jsou pro vývoj nezbytné nebo velmi ulehčí jeho vývoj.

Kapitola třetí popisuje funkční a nefunkční požadavky klienta, které definovaly, co by měl systém umět, podporovat, jak by měl fungovat a jaké vlastnosti a omezení bude. Pomocí strukturované analýzy bylo potřeba si udělat vlastní visuální návrh řešení, který byl popsán pomocí diagramu případů užití a diagramu aktivit. Předběžný návrh NoSQL databázového modelu byl graficky vyjádřen pomocí diagramu tříd. V této kapitole je popsaná také struktura uživatelské části a je zde návrh adres pro REST API. V poslední části třetí kapitoly je zobrazen grafický layout, jak by měla stránka ve výsledku vypadat a co by měla obsahovat.

Samotná realizace a implementace moderní webové aplikace běžící v reálném čase je popsána ve čtvrté kapitole. Je zde popsána horizontální a vertikální adresářová struktura, která je při vývoji využita a demonstruje flexibilitu MEAN stacku. Je zde zavedena konvence pojmenování, jelikož MVC architektura Express frameworku i AngularJS je velmi podobná a bez správných předem daných konvencí by často docházelo ke špatné orientaci a celkovému chaosu při procházení JavaScriptových souborů. Dále je zde popsána inicializace a konfigurace Express frameworku na straně serveru jak vykresluje své pohledy pomocí EJS enginu. Je zobrazena manipulace s modulem mongoose při CRUD operacích a jak Express

tyto manipulace využívá při využití RESTful API. Je zde popsán velmi užitečný middleware Passport pro implementaci přihlašování a poslední větší sekcí je samotná implementace frontendového frameworku AngulaJS, popis jeho MVC částí, služeb a modulu ngResource, který je nezbytný pro REST komunikaci mezi klientem a serverem. Poslední částí je pak využití modulu Socket.io, který pomocí zachytávání událostí umožňuje klientovi pracovat s webovou aplikací v reálném čase.

Tato práce také zkoumá běžné situace a problémy, kterými je nutné se při implementaci zabývat a poskytuje komplexní pohled na mnoho aspektů správného návrhu API s důrazem na využití správných postupů, které v maximální míře odpovídají REST principům a možnostem použitého protokolu HTTP.

Single-page aplikace realizována v této diplomové práci využívá MEAN stacku, který výrazně usnadnil její vývoj a umožňuje tak uživatelům využívat moderních elementů, především pak rychlost při načítání stránek, které ocení především uživatelé mobilních zařízení. Protože frontend i backend vychází z architektury MVC, který izoluje logiku od uživatelského rozhraní, tak je možné upravovat jednotlivé části odděleně. Díky tomu je možné aplikaci relativně jednoduše rozšířit a nabídnout nové funkce, případně kompletně změnit vzhled. Také díky tomu, že Express nabízí námi vytvořené API, tak na je možné vývoj na backendu zcela oddělit od vývoje frontendu aniž by to mělo jakýkoliv dopad na stávající funkcionalitu aplikace.

I když je implementovaná aplikace zcela funkční, nebyla při dokončení této práce stále nasazena do produkčního prostředí. Je to z důvodu stále probíhajících řešení projektu, jako je název URL adresy, na které se aplikace bude nacházet, a především problém při hostingu. V České republice aktuálně existuje pouze jediný hosting, u kterého je možné Node.js provozovat. Jiným řešením je využít známého zahraničního hostingu Heroku, kde je však třeba vyřešit všechny požadavky na funkcionality, které aplikace má. Do budoucna by bylo vhodné zajistit také lepší SEO optimalizaci pro vyhledavače, jelikož SPA aplikace jsou stále problémem v této oblasti.

6. Citovaná literatura

a) Odborná kniha (monografie, vysokoškolská učebnice, apod.)

BINDER, Robert V. *Testing object-oriented systems: models, patterns, and tools*. Boston: Addison-Wesley, 2009. ISBN 978-032-1700-674.

BOOCH, G. *Unified modeling language user guide*. Massachusetts: Addison-Wesley, 1999, 482 s. ISBN 02-015-7168-4.

BURKE, Bill. *RESTful Java with JAX-RS*. Sebastopol, Calif.: O'Reilly, 2011, xx, 288 p. ISBN 05-961-5804-1.

CANTELON, Mike, Marc HARTER, TJ HOLOWAYCHUK a Nathan RAJLICH. *Node.js in action*. New York, USA: Manning Publications, 2013, xx, 395 pages. ISBN 16-172-9057-2.

DAYLEY, Brad. *Node.js, MongoDB, and AngularJS Web Development*. Indiana, USA: Addison-Wesley Professional, 2014. 696 p. ISBN 978-0321995780.

FINK, Gil a Ido FLATOW. *Pro Single Page Application Development: Using Backbone.js and ASP.NET*. New York, USA: Apress, 2014. ISBN 978-1430266730.

FLANAGAN, David. *JavaScript: the definitive guide*. Fifth edition. Sebastopol: O´Reilly, 2006, 994 s. ISBN 05-961-0199-6. Dostupné z: http://my.safaribooksonline.com/0596101996/jscript5-CHP-1?portal=oreilly

GREEN, Brad a Shyam SESHADRI. *AngularJS*. First edition. Sebastopol, Kanada: O'Reilly Media, 2013, x, 183 pages. ISBN 978-144-9344-856.

LENGSTORF, Lengstorf a Phil LEGGETTER. *Realtime web apps: with HTML5, WebSocket, PHP, and jQuery*. Berkeley, Calif: Apress, 2013. ISBN 978-143-0246-206.

MIKOWSKI, Michael a Josh POWELL. *Single Page Web Applications: JavaScript end-to-end*. New York, USA: Manning Publications, 2013. ISBN 978-1617290756.

PLUGGE, Eelco, Peter MEMBREY a Tim HAWKINS. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. New York, USA: Springer-Verlag, 2010. ISBN 978-1-4302-3051-9.

ROHIT, Rai. *Socket.IO Real-time Web Application Development*. Velká Británie: Packt Publishing, 2013. 140 p. ISBN 978-1782160786.

SOMMERVILLE, Ian. *Software engineering*. 9th ed. Boston: Pearson, c2011, xv, 773 p. ISBN 978-013-7053-469.

VAISH, Gaurav. *Getting Started with NoSQL: [your guide to the world and technology of NoSQL]*. Online-Ausg. Packt Publishing: Packt Pub, 2013. ISBN 18-496-9498-2.

WEBBER, Jim, Savas PARASTATIDIS a Ian ROBINSON. *Rest in Practice*. Sebastopol, Kanada: O'Reilly Media, 2010. ISBN 978-0-596-80582-1.

WILSON, B. *Systems: concepts, methodologies, and applications*. 2nd ed. New York: Wiley, c1990, xvii, 391 p. ISBN 04-719-2716-3.

YOUNG, Ralph Rowland. *Effective requirements practices*. Boston, Ma.: Addison-Wesley, 2001, xxx, 359 p. ISBN 02-017-0912-0.

b) Článek v odborném časopise (periodiku) nebo ve sborníku z konference

FRANCE, R.B., S. GHOSH, T. DINH-TRONG a A. SOLBERG. Model-Driven Development Using UML 2.0: Promises and Pitfalls. In: *Computer* [online]. 2006, s. 35-55 [cit. 2015-02-28]. DOI: 10.1007/3-540-28554-7_3. Dostupné z: http://www.jot.fm/issues/issue_2009_05/article1.pdf

KUNDU, Debasish a Debasis SAMANTA. A Novel Approach to Generate Test Cases from UML Activity Diagrams. In: *The Journal of Object Technology* [online]. 2009 [cit. 2015-02-28]. DOI: 10.5381/jot.2009.8.3.a1. Dostupné z: http://www.jot.fm/issues/issue_2009_05/article1.pdf

LIE, Håkon Wium. *Cascading Style Sheets*. Norway, 2005. ISBN 1501-7710. Dostupné z: http://people.opera.com/howcome/2006/phd/#colophon. University of Oslo.

SUYONO, Hadi, Khalid Mohamed NOR, Sallehhudin YUSOF a Abdul Halim Abdul RASHID. Use-case and Sequence Diagram Models for Developing Transient Stability Software. In: *2006 IEEE International Power and Energy Conference* [online]. IEEE, 2006, s. 109-113 [cit. 2015-02-16]. ISBN 1-4244-0273-5. DOI: 10.1109/PECON.2006.346629. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4154473

TSUNEO NAKANISHI, Yasushi TSUCHIYA, Tadashi SAKAMOTO a Akira FUKUDA. Structured analysis for software product lines. In: *2009 IEEE 13th International Symposium on Consumer Electronics* [online]. IEEE, 2009, s. 915-919 [cit. 2015-02-16]. ISBN 978-1-

4244-2975-2. DOI: 10.1109/ISCE.2009.5157027. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5157027

VIDGEN, R. Requirements analysis and UML: use cases and class diagrams. In: *Computing and Control Engineering*. 2003, s. 12-17. ISSN 0956-3385. DOI: 10.1049/cce:20030202.

c) Elektronické dokumenty a ostatní

BELL, Donald. *UML basics: Part II: The activity diagram*. [online]. 2003 [cit. 2015-02-28]. Dostupné z: https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep03/f_umlbasics_db.pdf

BELL, Donald. UML basics: Part III: The class diagram. [online]. 2003 [cit. 2015-03-04]. Dostupné z: http://www.comscigate.com/uml/Basics/classDiagram.pdf

BERNERS-LEE, Tim. Hypertext Markup Language - 2.0. In: [online]. Network Working Group, 1995 [cit. 2015-03-09]. Dostupné z: http://tools.ietf.org/html/rfc1866

BUCKLER, Craig. W3C Announce HTML5 2014 Delivery Plan. In: [online]. 2012 [cit. 2015-03-10]. Dostupné z: http://www.sitepoint.com/w3c-html5-2014-plan/

CROCKFORD, Douglas. *Úvod do JSON* [online]. 2013 [cit. 2015-03-11]. Dostupné z: http://json.org/json-cz.html

DAHL, Ryan. *Node.js* [online]. 2015 [cit. 2015-04-14]. Dostupné z: https://nodejs.org/

GERCHEV, Ivaylo. *A Comprehensive Introduction to LESS* [online]. 2012 [cit. 2015-03-11]. Dostupné z: http://www.sitepoint.com/a-comprehensive-introduction-to-less/

GLOCK, Jonathan. *Node.js Framework Comparison: Express vs. Koa vs. Hapi* [online]. 2014 [cit. 2015-03-14]. Dostupné z: https://www.airpair.com/node.js/posts/nodejs-framework-comparison-express-koa-hapi

HAN, Sébastien. *Differences Between Active-active and Active-passive Cluster* [online]. 2012 [cit. 2015-03-14]. Dostupné z: http://www.sebastien-han.fr/blog/2012/05/26/differences-between-active-active-and-active-passive-cluster/

HARRISON, Christopher. *Introduction to Bootstrap – A Tutorial* [online]. 2014 [cit. 2015-03-11]. Dostupné z: https://www.edx.org/course/introduction-bootstrap-tutorial-microsoft-dev203x#.VQARhvyG8Ro

NIELSEN, Jakob. How Long Do Users Stay on Web Pages?. In: [online]. 2011 [cit. 2015-03-07]. Dostupné z: http://www.nngroup.com/articles/how-long-do-users-stay-on-web-pages/

O'GRADY, Stephen. *The RedMonk Programming Language Rankings: January 2015* [online]. 2015 [cit. 2015-03-12]. Dostupné z: http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

PATEL, Viral. *AngularJS: Introduction And Hello World Example* [online]. 2013 [cit. 2015-03-14]. Dostupné z: http://viralpatel.net/blogs/angularjs-introduction-hello-world-tutorial/

POTVIN, Cindy. *Introduction to LESS : an easier way to create CSS stylesheets* [online]. 2014 [cit. 2015-03-11]. Dostupné z: http://www.codeproject.com/Articles/724246/Introduction-to-LESS-an-easier-way-to-create-CSS-s

RICHARDSON, Deb. *A re-introduction to JavaScript (JS tutorial)* [online]. 2015 [cit. 2015-03-11]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

RUSSELL, Alex. *Comet: Low Latency Data for the Browser* [online]. 2006 [cit. 2015-03-16]. Dostupné z: http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/

SHTYLMAN, Roman. *Express.js Middleware Demystified* [online]. 2014 [cit. 2015-03-14]. Dostupné z: https://blog.safaribooksonline.com/2014/03/10/express-js-middleware-demystified/

SIKOS, Leslie. HTML5 Became a Standard, HTML 5.1 and HTML 5.2 on the Way. In: [online]. 2014 [cit. 2015-03-10]. Dostupné z: http://www.lesliesikos.com/html5-became-a-standard-html-5-1-and-html-5-2-on-the-way/

VASILE, Christian. HTML5 Introduction – What is HTML5 Capable of, Features, and Resources. In: [online]. 2013 [cit. 2015-03-10]. Dostupné z: http://www.1stwebdesigner.com/html5-introduction/

VOROSHILIN, Ivan. *Brewer's CAP Theorem Explained: BASE Versus ACID* [online]. 2012 [cit. 2015-03-14]. Dostupné z: http://ivoroshilin.com/2012/12/13/brewers-cap-theorem-explained-base-versus-acid/

ZALEWSKI, Michal. *Browser Security Handbook, part 2* [online]. 2009 [cit. 2015-03-16]. Dostupné z: https://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy

Seznam zkratek

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CERN	evropská organizace pro jaderný výzkum
CLI	Command-line interface
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
MVC	architektura Model–View–Controller
MVP	architektura Model–View–Presenter
MVW	architektura Model-View-Whatever
NoSQL	Not only SQL
NPM	Node.js Package Manager
ODM	Object Document Mapper
RDBMS	Relational Database Management Systém
REST	Representational State Transfer
SEO	Search Engine Optimization
SOAP	Simple Object Access Protocol
SPA	Single Page Application
SQL	Structured Query Language
SVG	Scalable Vector Graphics
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
XHTML	Extensible HyperText Markup Language
XHR	XMLHttpRequest
XML	Extensible Markup Language

Prohlášení o využití výsledků diplomové práce

Prohlašuji, že

- jsem byl(a) seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. – autorský zákon, zejména § 35 – užití díla v rámci občanských a náboženských obřadů, v rámci školních představení a užití díla školního a § 60 – školní dílo;
- beru na vědomí, že Vysoká škola báňská – Technická univerzita Ostrava (dále jen VŠB-TUO) má právo nevýdělečně, ke své vnitřní potřebě, diplomovou práci užít (§ 35 odst. 3);
- souhlasím s tím, že diplomová práce bude v elektronické podobě archivována v Ústřední knihovně VŠB-TUO a jeden výtisk bude uložen u vedoucího diplomové práce. Souhlasím s tím, že bibliografické údaje o diplomové práci budou zveřejněny v informačním systému VŠB-TUO;
- bylo sjednáno, že s VŠB-TUO, v případě zájmu z její strany, uzavřu licenční smlouvu s oprávněním užít dílo v rozsahu § 12 odst. 4 autorského zákona;
- bylo sjednáno, že užít své dílo, diplomovou práci, nebo poskytnout licenci k jejímu využití mohu jen se souhlasem VŠB-TUO, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly VŠB-TUO na vytvoření díla vynaloženy (až do jejich skutečné výše).

V Ostravě dne 25.4.2015

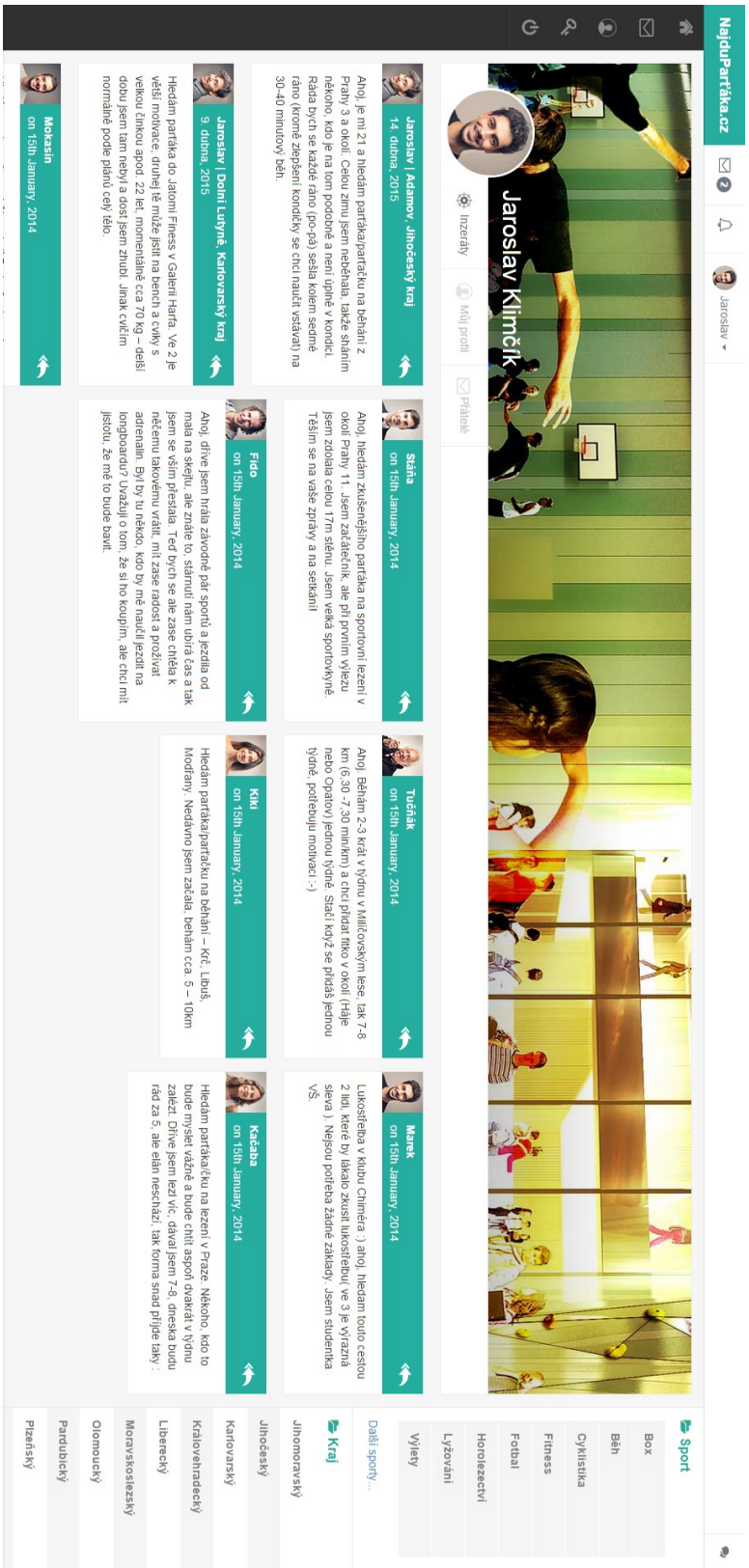
.....

Jaroslav Klimčík

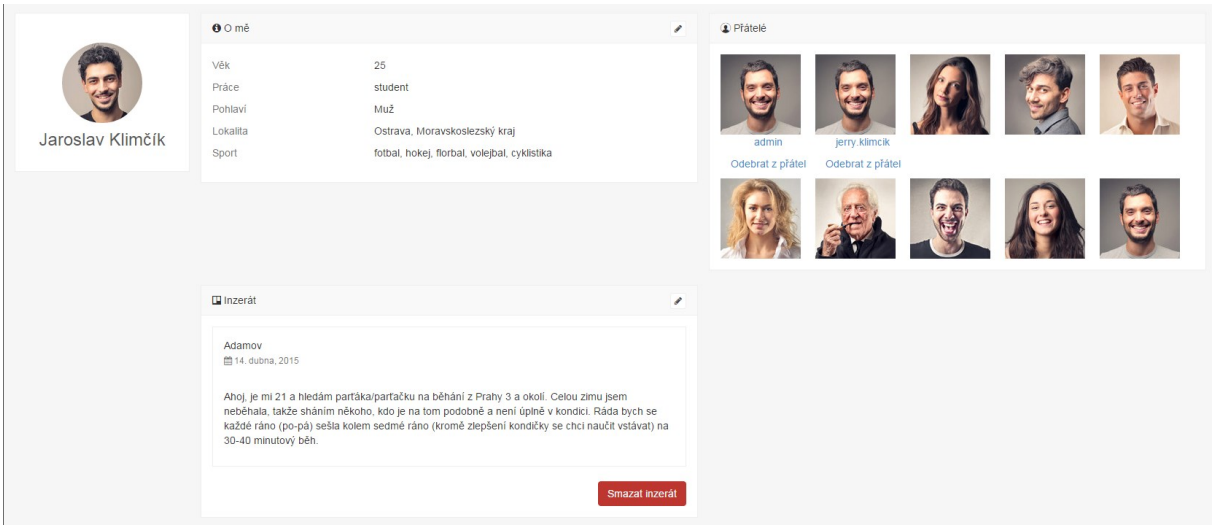
81

Seznam příloh

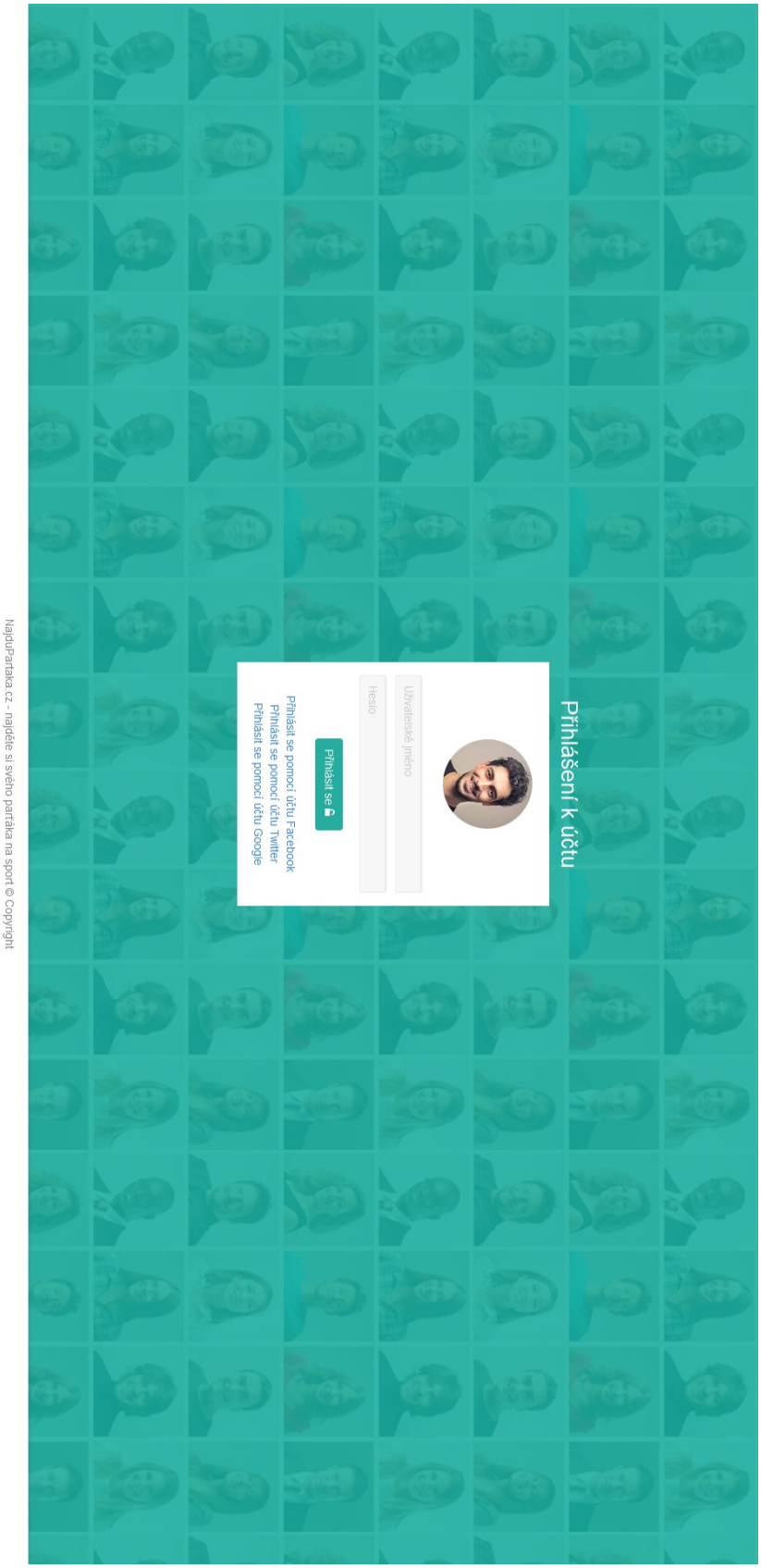
Příloha 1: Náhled stránky po přihlášení uživatele



Příloha č. 2 - Náhled profilu uživatele



Příloha č. 3 - Náhled přihlašovací stránky



Příloha č. 4 - Náhled administrace